November 2001
Vol. 11, No. 4

SLR Inside Natural

Tips, Techniques & Tricks for Natural

Boston & Darmstadt

FALL EDITION

Boston first

A bit of true humor first. I had developed Powerpoint presentations for Boston on my desktop PC. A few days before the Boston conference I put them on a diskette and transferred them to my laptop which would accompany me to Boston. No problems, I even had the diskette as a backup.

I had also developed a lot of programs under Natural 4.1.2 on my desktop. The night before driving up to the conference (No, this had nothing to do with a fear of flying; I always drive to Boston; takes me about 5 hours door to door, and I have my car there should I want to drive somewhere) I used SYSTRANS to unload the programs to a diskette, then transferred the programs to my laptop. Still no problems.

Then, the fun began. One of my presentations was devoted to using Natural 4/5 to develop event driven systems using Dialogs. When I tried to run the Dialogs from my desktop (Natural 4.1.2) on my laptop (Natural 4.1.1), they would not run. After a lot of cursing, I found myself reading the release notes for Version 4.1.2. They said that although all 4.1.1 structures would run under 4.1.2, the converse was not true. In particular, Dialogs were not backwards compatible.

I had several options. I could have gone searching for my 4.1.2 CD and upgraded my laptop to 4.1.2. I was tired (as SAG-L readers know, I am at my computer EARLY, 5:30 AM, and asleep early as well). The prospect of doing an install when I was quite tired was not very attractive. Second option. I had Version 5 on my laptop. However, this was a pre-release version. Did I want to risk giving a presentation using software I had not had time to play with? Not really. (ed note. Having now had some time to play with Version 5, this would have worked just fine. The pre release 5 seems to be quite bug free).

Just when I was about to pick one of the rather unattractive alternatives (probably the upgrade), I realized I had another alternative. I was, after all, driving up to Boston. I had a car that was only half filled with my "stuff" (brochures, etc). I had already planned to take my Desktop monitor to hook up to my laptop for display at my vendors booth. Why not just go all the way and take my desktop? Why not indeed?

In this Issue

Boston & Darmstadt

Inside Natural

Inside Natural published quarterly by:
S.L.ROBINSON & ASSOCIATES INC.
28 Teal Drive
Post Office Box L1235
Langhorne, Pennsylvania 19047

Telephone: -(215) 741-0820
Fax: ---------(215) 741-1351
E-mail -- SLRINC@AOL.COM
Web Page ----------------------
http://members.aol.com/slrinc

Domestic Subscriptions:
1 year - $ 95
2 years - $ 165
3 years - $ 225

Foreign Subscriptions:
Add $15 per year per subscription
or
Contact U.S. office for address/telephone of Regional Distributors

Subscriptions over ten from one company - 10% discount

Inside Natural is also distributed on magnetic media for unlimited copying
Subscription rate for this service is
$1,500 per year.
Back issues available : $25/issue

So there I was checking in at the Hyatt Hotel with all my "stuff" including a Desktop computer. Clearly I required the assistance of a bellhop to bring all this to my room. After the bellhop had put the computer on the desktop, and was about to pickup the large box containing my monitor, he turned to me and said, quite simply, "They make smaller computers now, called laptops". I could only chuckle to myself. Everyone is into computers these days. The night watchman in the Vendor area wanted to know where I got my screen saver (a memorial to the World Trade Center disaster). I made a copy for him; then we had a technical discussion regarding the pros/cons of switching to XP.

The Boston Conference had two major themes. The first was reflected by Software AG presentations. Version 5 of Natural is here, along with SpoD (Single Point of Development) which provides a really nice interface between PC's running Version 5 and mainframes running Version 3.1.5. There were quite a number of presentations on these topics which I deliberately skipped. Why?

The Boston conference is getting very robust. Most time slots had six or seven concurrent sessions. And that doesn't count the informal sessions that always seem to follow some interesting presentations (especially the late afternoon sessions which seem to run into pub time). Since I knew that some of the presentations would be repeated in Darmstadt, I skipped them in Boston. By the way, the breadth of the Boston Conference means that it is difficult for one person to "cover" the conference for a company. Many companies are sending multiple attendees all of whom are still being kept quite busy.

The other major theme from Boston was reflected by several user presentations, and a lot of informal sessions. The web is still a hot topic. Companies are still concerned with "putting it all on the web". Now a web "presence" is certainly valuable, the idea of putting all systems which have any exogenous links on the web is a bit extreme. Not all, indeed, perhaps, not many of a company's customers, are willing to give up "personal contact" in exchange for a web interface, even a well designed one. And, unfortunately, many websites are not terribly well designed. I have been extremely frustrated navigating the websites of some very well known, and large, companies. Nonetheless, webifying, as indicated by the sessions and the interest, is still the big topic these days.

Just a note, the Boston conference will be back in Boston next year, at the Hyatt, during the same time slot. Check occasionally at http://www.wizinc.com for updates. ❖

Now Darmstadt

The Darmstadt conference was the first of what will be known as the European Natural Programmers User Group (ENPUG), unless one of the proposed alternative names is adopted. The alternative names were suggested when it was apparent that attendees will come from Africa, Asia and the Middle East as well as Europe.

Since this was a mainly organizational meeting, there were no user presentations. Indeed, I presented the only session that was not presented by Software AG or one of its affiliates. Unlike Boston, which ENPUG will hopefully match in size after a few meetings, this initial meeting had but one track. Several of the presentations were the ones I deliberately skipped in Boston since I knew they would be repeated in Darmstadt. For example, I skipped the presentation on Natural Engineer in Boston, and caught it in Darmstadt. I did attend a couple of the Web enabling presentations in Boston, and saw a different one in Darmstadt.

I have always been a fan of Natural on the PC. I have worked with the entire sequence of products starting with Natural for Windows and NT, Natural New Dimension, Natural Lightstorm, and now Natural 4. I have found the development platform for Natural 4 to be quite a friendly one. Therefore I am quite intrigued by the possibilities of Natural 5 and SpoD. Being able to work with “mainframe Natural” from the PC Natural platform seems to be the best of both worlds. No more “extra software” between me and the mainframe. Goodbye to SYSTRANS. The new interface is quite programmer friendly. Next issue, after I have had a bit of time to play with the interface, I will have an indepth article based on my experiences.

There were three “techie” presentations in Darmstadt. You must understand what I consider a “techie” presentation. It is one in which the audience can participate since the presentation involves material they are familiar with. Even though the Version 5 and SpoD presentations contained technical material, it was new material for everyone in the audience, hence note taking was the order of the day, with few questions.

The three techie presentations were Andreas Schuetz presenting “Did you know; Seldom used or unknown features in Natural”, Thomas Frischmann (who has been with the Natural development team almost as long as Andreas) presenting “How to get performance using the right Natural statements” and my presentation on the Recorder and the Debugger. I will explore some of the topics from Thomas and Andreas’s presentations in this issue.

The next ENPUG meeting is tentatively scheduled for April in either Zurich or Geneva. For more information, send an e-mail to Dieter Klanke at dieter.klanke@softwareag.com ❖

Technical Potpourri

The scheduling for the Darmstadt conference was rather tight. Most of the sessions were scheduled for an hour or so. Thomas Frischmann had even less than that by the time his presentation started in the last time slot of the conference (someone, okay, me, had taken more than their allotted time). Thomas had over thirty items to discuss in less than an hour. He did an excellent job in getting through the presentation within the allotted time. However, I thought that some of the topics really could use a bit more time. So, I decided to take some of his topics and expand them here.

Some of the topics are fairly well known, others a bit obscure. All have the potential to improve performance of your systems. I suggest you take a look at them and consider how they would apply to your environment. I know that I, in the process of preparing this column, was more than a bit surprised by how costly certain “errors” could be. ❖

Natur Teknik

The “price” of System Variables and DECIDE

It is interesting how one topic of discussion can quickly lead to another. Sometimes, as in this case, the “spinoff” topic ends up being more interesting (at least to me) than the original topic.

One of Thomas’s performance tips concerned the use of System Variables. I remember hearing, years ago, probably from Andreas Schuetz, that accessing System Variables was fairly expensive. Why? System Variables are not stored “local” to your program. They are stored in various Natural control structures. Accessing them requires that Natural run special access modules. This is expensive. Okay, not if you just do it once or twice. Repeated access however, can be expensive. In the second half of this article, we will take a look at just how expensive this can be.

Now, however, we will look at something else. The example that Thomas used to show how expensive accessing System Variables can be was:

```
                                    MOVE *PF-KEY TO #PF-KEY
DECIDE ON FIRST VALUE OF *PF-KEY    DECIDE ON FIRST VALUE OF #PF-KEY
VALUE ‘PF1’..                                 VALUE ‘PF1’..
VALUE ‘PF2’..                                 VALUE ‘PF2’..
::::                                          ::::
VALUE ‘PF12’..                                VALUE ‘PF12’..
NONE VALUE ..                                 NONE VALUE ..
END-DECIDE                                    END-DECIDE
```

Why, I asked myself, should this be a problem? Natural is certainly smart enough, when executing the code on the left, to only obtain *PF-KEY once. I asked Thomas about this and he said that Natural did indeed obtain *PF-KEY repeatedly for the code shown. Andreas Schuetz was sitting just a few chairs to my left. Since I was sitting in the back row, it was easy to get up, slide to my left a few chairs and whisper my question to Andreas. Well, said Andreas, a DECIDE is just a series of IF statements. So, Natural probably repeats the acquisition of *PF-KEY.

“Why does it do that?”, I asked. If this were a DECIDE FOR statement, with potentially many variables involved in the various clauses, I could understand the distinct IF statements. But this is a DECIDE ON. There is only one variable involved in a DECIDE ON. Clearly Natural must be smart enough not to get the variable more than once.

I guess my handwaving, combined with the fact that it was getting on towards beer time, had an effect, because Andreas wavered at this point. Maybe there was a difference between the two DECIDE’s. We waited until the end of Thomas’s talk at which point Andreas was able to switch to German to translate my concerns to Thomas. Andreas’s animated translation of my observation did not sway Thomas. It is always a series of IF’s, Thomas said. I was still unconvinced. Ah, the arrogance of the ignorant.

Investigation would have to wait awhile however. Thomas’s talk was the last of the conference. Then it was off for a quick beer followed by a 30 minute drive to a fine goose dinner replete with more beer, excellent wine, and, for anyone who could remotely claim to be sober, grappa after dessert. Now I understand why we took taxis to the restaurant rather than driving. Needless to say, the DECIDE command was not discussed during dinner, at least not that I remember.

It was not until I was on a plane, flying home the following day that I was able to write some code and begin playing with the DECIDE commands. It occurred to me that if a DECIDE ON is indeed treated like a series of IF statements, you could get some pretty strange results. At least here, I was correct. Consider the following program. I have avoided using System Variables for the moment. No sense confusing two issues in one program.

```
* THIS PROGRAM DEMONSTRATES A POTENTIAL
* PROBLEM WITH THE DECIDE COMMAND
*
DEFINE DATA LOCAL
1 #A (N3) INIT <1>
END-DEFINE
*
INCLUDE AATITLER
INCLUDE AASETC
*
DECIDE ON EVERY VALUES   #A
VALUE 1
  WRITE 5T '#A IS 1'
  ADD 1 TO #A
VALUE 2
  WRITE 5T '#A IS 2'
  ADD 1 TO #A
VALUE 3
  WRITE 5T '#A IS 3'
ANY VALUE
  WRITE 5T 'THIS SHOULD BE PRINTED (ANY VALUE)'
ALL VALUES
  WRITE 5T 'CLEARLY THIS CANNOT HAPPEN (ALL VALUES)'
NONE VALUE
  WRITE 5T 'CLEARLY THIS WILL NOT HAPPEN'
END-DECIDE
*
END
```

Okay, the program is a bit silly. Now look at the output.

```
PAGE #   1                   DATE:    Nov 19, 2001
PROGRAM: DECIDE01            LIBRARY: INSIDE

#A IS 1
#A IS 2
#A IS 3
THIS SHOULD BE PRINTED
CLEARLY THIS CANNOT HAPPEN
```

Okay, lets take a look at the output. Actually, you don't have to look too hard to see the problem. Since the DECIDE command is basically a series of IF statements, and we change #A within the DECIDE clauses, we end up with the rather strange pronouncement that #A is 1,2, and 3.

Okay, you are saying this was a silly program. Here is a more realistic piece of code that suffers from the same problem.

```
* THIS PROGRAM DEMONSTRATES A POTENTIAL PROBLEM
* WHEN USING *PF-KEY IN A DECIDE COMMAND
*
INCLUDE AATITLER
INCLUDE AASETC
*
SET KEY PF1 PF2 PF3 PF4
*
INPUT 3/10 'THIS WOULD BE A LIST OF ACTIONS FOR'
    / 10T 'DIFFERENT PF KEYS. FOR DEMONSTRATION'
    / 10T 'PRESS PF2.'
*
DECIDE ON EVERY VALUE OF *PF-KEY
  VALUE 'PF1'    IGNORE
  VALUE 'PF2'    PERFORM WAS-PF2
  VALUE 'PF3'    WRITE 5T 'YES, DESTROY MY DATABASE'
  VALUE 'PF4'    IGNORE
  NONE VALUE     REINPUT 'PRESS A VALID PF KEY'
END-DECIDE
*
DEFINE SUBROUTINE WAS-PF2
INPUT 3/10 'THIS MIGHT BE SOME SORT OF CONFIRMATION'
    / 10T 'WINDOW FOR PF2. SUPPOSE (OKAY, IT WOULD'
    / 10T 'BE POOR DESIGN) THAT YOU CONFIRM BY '
    / 10T 'PRESSING PF3.. PRESS THIS KEY (PF3)'
*
* MISCELLANEOUS CODE
END-SUBROUTINE
*
END
```

The code is not that unusual. Assume PF2 activates some sort of UPDATE or DELETE command. Our little subroutine performs a common function, a confirmation window. I followed the directions in the code. I first pressed PF2. Then, when prompted by the confirmation window, I pressed PF3. Here is the output that follows.

```
PAGE #   1                   DATE:    Nov 20, 2001
PROGRAM: DECIDE02            LIBRARY: INSIDE

YES, DESTROY MY DATABASE
```

Okay, if I went to all the trouble to require confirmation for a simple UPDATE or DELETE, I certainly will have a second chance to avoid destroying the database. The point is that in this program, like the earlier program, DECIDE01, the value of *PF-KEY can change during the processing of the DECIDE command.

What this means is you have to be very careful how you code your DECIDE commands. For example, a DECIDE ON for *PF-KEY should probably be coded (unlike in DECIDE02 above) as DECIDE....FIRST. However, to be fair, there is often occasion to not use FIRST. I have seen many programs that employ a PF key to indicate an UPDATE or a DELETE. Then there would be a VALUE 'PF5', 'PF8' END TRANSACTION. That is, if we did either an update or delete, we would issue an END TRANSACTION. Here, FIRST would not be appropriate.

System Variables

Now that we have seen a potential problem with DECIDE commands, it is time to look at the other part of this discussion, namely, the “cost” of accessing System Variables repeatedly. In the following program we will contrast the time for two approaches to testing the value of a System Variable, namely *PF-KEY.

```
0010 DEFINE DATA LOCAL
0020 1 #LOOP (P5)
0030 1 #HOLDER (A4)
0040 END-DEFINE
0050 *
0060 INCLUDE AATITLER
0070 INCLUDE AASETC
0080 *
0090 SETA. SETTIME
0100 FOR #LOOP = 1 TO 5000
0110 DECIDE ON FIRST VALUE *PF-KEY
0120 VALUE 'PF6' IGNORE
0130 VALUE 'PF7' IGNORE
0140 VALUE 'PF8' IGNORE
0150 VALUE 'PF9' IGNORE
0160 NONE IGNORE
0170 END-DECIDE
0180 END-FOR
0190 WRITE '*PF-KEY TIME' *TIMD (SETA.)
0200 SETB. SETTIME
0210 FOR #LOOP = 1 TO 5000
0220 *
0230 MOVE *PF-KEY TO #HOLDER
0240 *
0250 DECIDE ON FIRST VALUE #HOLDER
0260 VALUE 'PF6' IGNORE
0270 VALUE 'PF7' IGNORE
0280 VALUE 'PF8' IGNORE
0290 VALUE 'PF9' IGNORE
0300 NONE IGNORE
0310 END-DECIDE
0320 END-FOR
0330 WRITE '#HOLDER TIME' *TIMD (SETB.)
0340 END
```

Note that in our first loop, we directly test *PF-KEY. As we noted above, this basically means we will be obtaining *PF-KEY four times, once for each VALUE clause. By contrast, in our second loop we are obtaining *PF-KEY just once for each iteration of the loop. The value is then placed in #HOLDER for testing by the DECIDE statement.

```
PAGE #   1                  DATE:    Nov 15, 2001
PROGRAM: SYSVAR01           LIBRARY: INSIDE

*PF-KEY TIME       8
#HOLDER TIME       3
```

Take a look at the times. More than double for the *PF-KEY loop. Think that’s bad? I have seen many programs that use all the PF keys. Here is a timing with 12 PF keys (yes, I know, there are more than 12 PF keys. As a matter of design principle, I try to avoid using more than the first twelve. Actually, I try to use even fewer keys for any given screen. My experience is that error rates increase markedly beyond three of four).

```
0010 DEFINE DATA LOCAL
0020 1 #LOOP (P5)
0030 1 #HOLDER (A4)
0040 END-DEFINE
0050 *
0060 INCLUDE AATITLER
0070 INCLUDE AASETC
0080 *
0090 SETA. SETTIME
0100 FOR #LOOP = 1 TO 5000
0110 DECIDE ON FIRST VALUE *PF-KEY
0120 VALUE 'PF1' IGNORE
0130 VALUE 'PF2' IGNORE
0140 VALUE 'PF3' IGNORE
0150 VALUE 'PF4' IGNORE
0160 VALUE 'PF5' IGNORE
0170 VALUE 'PF6' IGNORE
0180 VALUE 'PF7' IGNORE
0190 VALUE 'PF8' IGNORE
0200 VALUE 'PF9' IGNORE
0210 VALUE 'PF10' IGNORE
0220 VALUE 'PF11' IGNORE
0230 VALUE 'PF12' IGNORE
0240 NONE IGNORE
0250 END-DECIDE
0260 END-FOR
0270 WRITE 5T '*PF-KEY TIME' *TIMD (SETA.)
0280 *
0290 SETB. SETTIME
0300 FOR #LOOP = 1 TO 5000
0310 *
0320 MOVE *PF-KEY TO #HOLDER
0330 *
0340 DECIDE ON FIRST VALUE #HOLDER
0350 VALUE 'PF1' IGNORE
0360 VALUE 'PF2' IGNORE
0370 VALUE 'PF3' IGNORE
0380 VALUE 'PF4' IGNORE
0390 VALUE 'PF5' IGNORE
0400 VALUE 'PF6' IGNORE
0410 VALUE 'PF7' IGNORE
0420 VALUE 'PF8' IGNORE
0430 VALUE 'PF9' IGNORE
0440 VALUE 'PF10' IGNORE
0450 VALUE 'PF11' IGNORE
0460 VALUE 'PF12' IGNORE
0470 NONE IGNORE
0480 END-DECIDE
0490 END-FOR
0500 WRITE 5T '#HOLDER TIME' *TIMD (SETB.)
0510 *
0520 SETC. SETTIME
0530 FOR #LOOP = 1 TO 5000
0540 IGNORE
0550 END-FOR
0560 *
0570 WRITE 5T 'FOR LOOP TIME' *TIMD (SETC.)
0580 END
```

And the rather emphatic output.

```
PAGE #   1                    DATE:    Nov 25, 2001
PROGRAM: SYSVAR1X             LIBRARY: INSIDE

*PF-KEY TIME       23
#HOLDER TIME        3

FOR LOOP TIME                1
```

Here are some numbers to seriously consider. I “threw in” the FOR loop, since that is “overhead” for both loops. Subtracting that from each loop time, the relevant times are 22 and 2; a factor of eleven. This simple coding of a DECIDE for *PF-KEY is that inefficient.

Of course, you probably would not have an *PF-KEY test in a loop. It would most likely be used in an online system following an INPUT statement. HOWEVER, the potential cost of accessing a System Variable, any System Variable (with the exception of “loop specific” variables like *NUMBER, *COUNTER, and *ISN which are stored locally), repeatedly should warrant careful attention.

Perhaps the most “abused” of the System Variables would be one of the date variables, for example, *DATX. I must confess, I have been (note past tense; this will not happen again) as guilty as anyone of writing code like:

```
READ ORDERS
IF *DATX - ORDER-DATE GT 5
         :::
         END-IF
```

Of course, what I should do is a MOVE *DATX TO #DATE somewhere before the READ loop, then do an IF #DATE - ORDER-DATE... What does this error cost?
Here is a simple program.

```
DEFINE DATA LOCAL
1 #LOOP (P7)
1 #HOLDER (A4)
1 #DATE-1 (D)
1 #DATE-2 (D)
END-DEFINE
*
INCLUDE AATITLER
INCLUDE AASETC
*
MOVE *DATX TO #DATE-1 #DATE-2
SUBTRACT 3 FROM #DATE-2
SETA. SETTIME
FOR #LOOP = 1 TO 150000
IF *DATX - #DATE-2 GT 3
  IGNORE
  END-IF
END-FOR
WRITE 5T '*DATX TIME' *TIMD (SETA.)
*
SETB. SETTIME
FOR #LOOP = 1 TO 150000
IF #DATE-1 - #DATE-2 GT 3
   IGNORE
   END-IF
END-FOR
WRITE 5T 'LOCAL VARIABLE TIME' *TIMD (SETB.)
*
SETC. SETTIME
FOR #LOOP = 1 TO 150000
   IGNORE
END-FOR
WRITE 5T 'FOR LOOP TIME' *TIMD (SETC.)
END
```

And the rather interesting output.

```
PAGE #   1                    DATE:    Nov 26, 2001
PROGRAM: SYSVAR04             LIBRARY: INSIDE

*DATX TIME       29
LOCAL VARIABLE TIME       19
FOR LOOP TIME       8
```

Subtracting the common FOR loop time of eight from the other two times, we are looking at 21 versus 11. It is almost twice as expensive to keep getting *DATX as it is to get it once, move it to a local variable, and thereafter, reference the local variable. Okay, for the sticklers out there. It is of course possible for the date to change during the running of a program. Make sure this will not be a problem for your application before switching to the local variable logic.

A Surprise

As most readers know, it does not take much to send me off on a tangent exploring some strange aspect of Natural. It bothered me that the DECIDE command retrieved *PF-KEY once per VALUE clause. I decided (no pun intended) to play a bit. How about an IF statement with a bunch of OR's?

```
0010 DEFINE DATA LOCAL
0020 1 #LOOP (P5)
0030 1 #HOLDER (A4)
0040 END-DEFINE
0050 *
0060 INCLUDE AATITLER
0070 INCLUDE AASETC
0080 *
0090 SETA. SETTIME
0100 FOR #LOOP = 1 TO 15000
0110 IF *PF-KEY = 'PF6' OR = 'PF7' OR = 'PF8' OR = 'PF9'
0120 IGNORE
0130 ELSE
0140 IGNORE
0150 END-IF
0160 END-FOR
0170 WRITE 'IF TIME' *TIMD (SETA.)
0180 *
0190 SETB. SETTIME
0200 FOR #LOOP = 1 TO 15000
0210 DECIDE ON FIRST VALUE *PF-KEY
0220 VALUE 'PF6' IGNORE
0230 VALUE 'PF7' IGNORE
0240 VALUE 'PF8' IGNORE
0250 VALUE 'PF9' IGNORE
0260 NONE IGNORE
0270 END-DECIDE
0280 END-FOR
0290 WRITE 'DECIDE TIME' *TIMD (SETB.)
0300 END
```

And our rather surprising output.

```
PAGE #   1                    DATE:    Nov 21, 2001
PROGRAM: SYSVAR03             LIBRARY: INSIDE

IF TIME        7
DECIDE TIME       24
```

I thought this was rather interesting so I immediately sent an e-mail to the Natural development team. Back came the answer, the IF, unlike the DECIDE, will only acquire the value of *PF-KEY once.

Well, this was interesting. Suppose I was not referencing a System Variable, just a plain old local variable.

```
0010 DEFINE DATA LOCAL
0020 1 #LOOP (P7)
0030 1 #HOLDER (A4)
0040 END-DEFINE
0050 *
0060 INCLUDE AATITLER
0070 INCLUDE AASETC
0080 *
0090 SETA. SETTIME
0100 FOR #LOOP = 1 TO 150000
0110 IF #HOLDER = 'PF6' OR = 'PF7' OR = 'PF8' OR = 'PF9'
0120 IGNORE
0130 ELSE
0140 IGNORE
0150 END-IF
0160 END-FOR
0170 WRITE 'IF TIME' *TIMD (SETA.)
0180 *
0190 SETB. SETTIME
0200 FOR #LOOP = 1 TO 150000
0210 DECIDE ON FIRST VALUE #HOLDER
0220 VALUE 'PF6' IGNORE
0230 VALUE 'PF7' IGNORE
0240 VALUE 'PF8' IGNORE
0250 VALUE 'PF9' IGNORE
0260 NONE IGNORE
0270 END-DECIDE
0280 END-FOR
0290 WRITE 'DECIDE TIME' *TIMD (SETB.)
0300 END
```

And the rather expected output.

```
    PAGE #   1                    DATE:    Nov 26, 2001
    PROGRAM: SYSVAR02             LIBRARY: INSIDE

IF TIME       22
DECIDE TIME       27
```

SO, clearly, if I wish to know whether a variable has one of several values, the IF is a lot more efficient than the DECIDE.

I will admit that I have always used IFs rather than DECIDEs for such a requirement. Without ever quantifying the difference, I have always used IF rather than DECIDE unless I would be making use of the “power” of DECIDE. For me, the “power” of a DECIDE resides in its optional clauses; ANY and ALL and its required clause NONE.

To duplicate the role of these clauses, while using IF statements, I have to use a logical variable or a counter. This makes the code considerably harder to follow for someone other than the original coder, hence I opt for the DECIDE. However, in light of the above, I decided to quantify the difference between IF and DECIDE for several different scenarios. In this first test, none of the conditions are met.

```
* THIS PROGRAM DEMONSTRATES A TIMING
* COMPARISON BETWEEN IF'S AND DECIDE COMMANDS
*
DEFINE DATA LOCAL
1 #A (N3) INIT <5>
1 #FLAG (L)
1 #LOOP (P7)
END-DEFINE
*
INCLUDE AATITLER
INCLUDE AASETC
*
SETA. SETTIME
FOR #LOOP = 1 TO 150000
DECIDE ON EVERY VALUES   #A
VALUE 1,3
  IGNORE
VALUE 2,4
  IGNORE
ANY VALUE
  WRITE 'THIS SHOULD NOT BE PRINTED (ANY VALUE)'
NONE VALUE
  IGNORE
END-DECIDE
END-FOR
WRITE 5T 'DECIDE TIME' *TIMD (SETA.)
*
SETB. SETTIME
FOR #LOOP = 1 TO 150000
*
IF #A = 1 OR = 3
  MOVE TRUE TO #FLAG
  END-IF
IF #A = 2 OR = 4
  MOVE TRUE TO #FLAG
  END-IF
IF #FLAG = TRUE
  IGNORE
  ELSE
  IGNORE
  END-IF
END-FOR
WRITE 5T 'IF TIME' *TIMD (SETB.)
*
END
```

And our output:

```
PAGE #   1                  DATE:    Nov 26, 2001
PROGRAM: DECIDE03           LIBRARY: INSIDE

DECIDE TIME       35
IF TIME       31
```

As expected, the DECIDE command is still more expensive than the IF statements. However, the difference is fairly small, about 10%. I was curious whether the difference would be any different if one of the conditions were met. Basically, this would add the cost of a MOVE TRUE... to the IF loop and the counterpart cost for the DECIDE, which is probably the same sort of flag setting.

```
* THIS PROGRAM DEMONSTRATES A TIMING
* COMPARISON BETWEEN IF'S AND DECIDE COMMANDS
*
DEFINE DATA LOCAL
1 #A (N3) INIT <2>
1 #FLAG (L)
1 #LOOP (P7)
END-DEFINE
*
INCLUDE AATITLER
INCLUDE AASETC
*
SETA. SETTIME
FOR #LOOP = 1 TO 150000
DECIDE ON EVERY VALUES   #A
VALUE 1,3
  IGNORE
VALUE 2,4
  IGNORE
ANY VALUE
  IGNORE
NONE VALUE
  IGNORE
END-DECIDE
END-FOR
WRITE 5T 'DECIDE TIME' *TIMD (SETA.)
*
SETB. SETTIME
FOR #LOOP = 1 TO 150000
*
IF #A = 1 OR = 3
  MOVE TRUE TO #FLAG
  END-IF
IF #A = 2 OR = 4
  MOVE TRUE TO #FLAG
  END-IF
IF #FLAG = TRUE
  IGNORE
  ELSE
  IGNORE
  END-IF
END-FOR
WRITE 5T 'IF TIME' *TIMD (SETB.)
*
END
```

And our timings.

```
PAGE #   1                  DATE:    Nov 26, 2001
PROGRAM: DECIDE04           LIBRARY: INSIDE

DECIDE TIME       31
IF TIME       28
```

Now this is rather interesting. It shows how you can mentally go down one path and fail to understand what might happen. I expected that the timings for DECIDE04 would be greater than those for DECIDE03. Why? As noted above, I expected the extra MOVE TRUE TO #FLAG for the IF code, and the counterpart code for the DECIDE, would result in slight increases to the times, not the slight decreases shown above.

What did I forget? Take a look at the IF #A = 2 OR = 4. Since #A = 2, Natural is smart enough not to bother doing the test for = 4. A similar savings can be expected for the DECIDE clause VALUE 2,4. At least that was my guess based on looking at the code. I changed the INIT value of #A to 4 rather than 2, and re-ran the program. Here are the results:

```
PAGE #   1                    DATE:    Nov 26, 2001
PROGRAM: DECIDE05             LIBRARY: INSIDE

DECIDE TIME       36
IF TIME       32
```

Note the slight increase (36 vs 35 and 32 vs 31) compared with DECIDE03 above. This would be the "cost" of the additional MOVE TRUE TO #A's and the counterpart internal operation for the DECIDE.

Summary

Some "absolutes" first. Retrieving System Variables is EXPENSIVE. Try to avoid repetitive retrievals of the same value (e.g. *DATX). Instead, retrieve the System Variable once and store it in a local variable. Thereafter, use the local variable.

Be careful with DECIDE commands. They are indeed independent, serial, IF statements. Changing a variable in the DECIDE can result in "strange" results. This is easier to have happen with DECIDE FOR's than DECIDE ON's (more variables might be involved).

IF's with OR's are more efficient than DECIDEs at ascertaining if a variable has one of several values. Of course, if you also need to know "which value", it is probably better (and more efficient) to use a DECIDE with a VALUE clause for each value you are searching for. The ANY clause will get you the OR condition and the individual VALUE clauses will identify the specific value. ❖

Arrays, Scalars, and SUBSTRING

Okay, you have an idea from the last article about the theme of Thomas's talk. It was mainly concerned with performance. What surprised me was how many programmers are not aware of some of the simple ways to improve Natural performance.

Consider arrays. Which should be faster, a reference to something like #ARRAY (3) or a reference to something like #ARRAY (#INDEX) ? "Clearly" the constant three should be faster (we will see later that this is NOT true). Yet I see people writing code like:

```
IF some criteria
        MOVE 2 TO #SUB
        ELSE
        MOVE 3 TO #SUB
        END-IF
MOVE #ARRAY (#SUB) TO #PROCESS
```

What does this cost in terms of performance. Here is a rather simple example:

```
* THIS PROGRAM SHOWS THE SIGNIFICANCE OF A SEEMINGLY
* UNIMPORTANT PROGRAMMING DECISION REGARDING ARRAYS
*
DEFINE DATA LOCAL
1 #ARRAY (A5/1:5)
1 #INDEX (I4)
1 #TARGET (A5)
1 #CRITERIA (A3)
1 #LOOP (P7)
END-DEFINE
*
INCLUDE AATITLER
INCLUDE AASETC
*
SETA. SETTIME
FOR #LOOP = 1 TO 150000
IF #CRITERIA = ' '
   COMPUTE #INDEX = 2
   ELSE
   COMPUTE #INDEX = 3
   END-IF
MOVE #ARRAY (#INDEX) TO #TARGET
END-FOR
WRITE 5T 'VARIABLE SUBSCRIPT TIME' *TIMD (SETA.)
*
SETB. SETTIME
FOR #LOOP = 1 TO 150000
IF #CRITERIA = ' '
   MOVE #ARRAY (2) TO #TARGET
   ELSE
   MOVE #ARRAY (3) TO #TARGET
   END-IF
END-FOR
WRITE 5T 'FIXED SUBSCRIPT TIME' *TIMD (SETB.)
END
```

```
PAGE #   1                    DATE:    Nov 30, 2001
PROGRAM: ARRAY04              LIBRARY: INSIDE

VARIABLE SUBSCRIPT TIME       18
FIXED SUBSCRIPT TIME       16
```

A ten percent improvement in performance, merely by writing better code. Note that the real difference here is the elimination of the “extra” COMPUTE statement, not the difference between a fixed and a variable subscript (to repeat, more about this later).

In general, referencing a specific variable is considerably more efficient than referencing an array. Why? Consider all the work Natural must do for any array reference. First, Natural must “locate” the specified array member. Even a constant subscript, like the number 3, requires a computation. A variable subscript like #INDEX requires even more computation. Then, Natural must do an “out of bounds” check, to ensure you are not trying to reference an occurrence that is beyond the declared extent of the array. All of this consumes CPU time. Here is an informative little program:

```
0010 * THIS PROGRAM SHOWS THE SIGNIFICANCE OF A SEEMINGLY
0020 * UNIMPORTANT PROGRAMMING DECISION REGARDING ARRAYS
0030 *
0040 DEFINE DATA LOCAL
0050 1 #ARRAY (A5/1:5)
0060 1 REDEFINE #ARRAY
0070   2 FILLER 5X
0080   2 #SECOND (A5)
0090   2 #THIRD (A5)
0100 1 #INDEX (I4)
0110 1 #TARGET (A5)
0120 1 #CRITERIA (A3)
0130 1 #LOOP (P7)
0140 END-DEFINE
0150 *
0160 INCLUDE AATITLER
0170 INCLUDE AASETC
0180 *
0190 SETA. SETTIME
0200 FOR #LOOP = 1 TO 150000
0210 IF #CRITERIA = ' '
0220    COMPUTE #INDEX = 2
0230    ELSE
0240    COMPUTE #INDEX = 3
0250    END-IF
0260 MOVE #ARRAY (#INDEX) TO #TARGET
0270 END-FOR
0280 WRITE 5T 'VARIABLE SUBSCRIPT TIME' *TIMD (SETA.)
0290 *
0300 SETB. SETTIME
0310 FOR #LOOP = 1 TO 150000
0320 IF #CRITERIA = ' '
0330    MOVE #ARRAY (2) TO #TARGET
0340    ELSE
0350    MOVE #ARRAY (3) TO #TARGET
0360    END-IF
0370 END-FOR
0380 WRITE 5T 'FIXED SUBSCRIPT TIME' *TIMD (SETB.)
0390 *
0400 SETC. SETTIME
0410 FOR #LOOP = 1 TO 150000
0420 IF #CRITERIA = ' '
0430    MOVE #SECOND TO #TARGET
0440    ELSE
0450    MOVE #THIRD TO #TARGET
0460    END-IF
0470 END-FOR
0480 WRITE 5T 'NON ARRAY TIME' *TIMD (SETC.)
0490
0500 END
```

And the expected output:

```
PAGE #   1                    DATE:    Nov 30, 2001
PROGRAM: ARRAY05              LIBRARY: INSIDE

VARIABLE SUBSCRIPT TIME       18
FIXED SUBSCRIPT TIME       16
NON ARRAY TIME       13
```

You may not have been impressed by the 10% savings of using fixed rather than variable subscripts. How about the 20% additional savings by using a simple variable reference (albeit a REDEFINE of an array) rather than a fixed subscript reference?

Not enough of an improvement. After all, there is only one reference per iteration. Suppose there were more? A lot more.

```
0010 * THIS PROGRAM SHOWS THE SIGNIFICANCE OF A SEEMINGLY
0020 * UNIMPORTANT PROGRAMMING DECISION REGARDING ARRAYS
0030 *
0040 DEFINE DATA LOCAL
0050 1 #ARRAY (A5/1:5)
0060 1 REDEFINE #ARRAY
0070   2 #FIRST (A5)
0080   2 #SECOND (A5)
0090   2 #THIRD (A5)
0100   2 #FOURTH (A5)
0110   2 #FIFTH (A5)
0120 1 #INDEX (I4)
0130 1 #TARGET (A5)
0140 1 #CRITERIA (A3)
0150 1 #LOOP (P7)
0160 END-DEFINE
0170 *
0180 INCLUDE AATITLER
0190 INCLUDE AASETC
0200 *
0210 SETB. SETTIME
0220 FOR #LOOP = 1 TO 150000
0230 IF #CRITERIA = ' '
0240    MOVE #ARRAY (1) TO #TARGET
0250    MOVE #ARRAY (2) TO #TARGET
0260    MOVE #ARRAY (3) TO #TARGET
0270    MOVE #ARRAY (4) TO #TARGET
0280    MOVE #ARRAY (5) TO #TARGET
0290    MOVE #ARRAY (1) TO #TARGET
0300    MOVE #ARRAY (2) TO #TARGET
0310    MOVE #ARRAY (3) TO #TARGET
0320    MOVE #ARRAY (4) TO #TARGET
0330    MOVE #ARRAY (5) TO #TARGET
0340    END-IF
0350 END-FOR
0360 WRITE 5T 'FIXED SUBSCRIPT TIME' *TIMD (SETB.)
```

```
0370 *
0380 SETC. SETTIME
0390 FOR #LOOP = 1 TO 150000
0400 IF #CRITERIA = ' '
0410    MOVE #FIRST TO #TARGET
0420    MOVE #SECOND TO #TARGET
0430    MOVE #THIRD TO #TARGET
0440    MOVE #FOURTH TO #TARGET
0450    MOVE #FIFTH TO #TARGET
0460    MOVE #FIRST TO #TARGET
0470    MOVE #SECOND TO #TARGET
0480    MOVE #THIRD TO #TARGET
0490    MOVE #FOURTH TO #TARGET
0500    MOVE #FIFTH TO #TARGET
0510    END-IF
0520 END-FOR
0530 WRITE 5T 'NON ARRAY TIME' *TIMD (SETC.)
0540
0550 END
```

And our output:

```
PAGE #   1                  DATE:    Nov 30, 2001
PROGRAM: ARRAY06            LIBRARY: INSIDE

FIXED SUBSCRIPT TIME       43
NON ARRAY TIME       22
```

Upping the number of references from one to ten certainly makes a difference. The simple variable reference is now half the subscript reference.

There is one other way to approach what would ordinarily be array references, SUBSTRING's. Many programmers seem to feel that SUBSTRING is the fastest way to do anything resembling array operations. WRONG!! Take a look at the following program and output.

```
0010 * THIS PROGRAM COMPARES SUBSTRING WITH REDEFINE
0020 *
0030 DEFINE DATA LOCAL
0040 1 #STRING (A25)
0050 1 REDEFINE #STRING
0060   2 #FIRST (A5)
0070   2 #SECOND (A5)
0080   2 #THIRD (A5)
0090   2 #FOURTH (A5)
0100   2 #FIFTH (A5)
0110 1 #INDEX (I4)
0120 1 #TARGET (A5)
0130 1 #CRITERIA (A3)
0140 1 #LOOP (P7)
0150 END-DEFINE
0160 *
0170 INCLUDE AATITLER
0180 INCLUDE AASETC
```

```
0190 *
0200 SETB. SETTIME
0210 FOR #LOOP = 1 TO 150000
0220 IF #CRITERIA = ' '
0230    MOVE SUBSTRING (#STRING,1,5) TO #TARGET
0240    MOVE SUBSTRING (#STRING,6,5) TO #TARGET
0250    MOVE SUBSTRING (#STRING,11,5) TO #TARGET
0260    MOVE SUBSTRING (#STRING,16,5) TO #TARGET
0270    MOVE SUBSTRING (#STRING,21,5) TO #TARGET
0280    MOVE SUBSTRING (#STRING,1,5) TO #TARGET
0290    MOVE SUBSTRING (#STRING,6,5) TO #TARGET
0300    MOVE SUBSTRING (#STRING,11,5) TO #TARGET
0310    MOVE SUBSTRING (#STRING,16,5) TO #TARGET
0320    MOVE SUBSTRING (#STRING,21,5) TO #TARGET
0330    END-IF
0340 END-FOR
0350 WRITE 5T 'SUBSTRING TIME' *TIMD (SETB.)
0360 *
0370 SETC. SETTIME
0380 FOR #LOOP = 1 TO 150000
0390 IF #CRITERIA = ' '
0400    MOVE #FIRST TO #TARGET
0410    MOVE #SECOND TO #TARGET
0420    MOVE #THIRD TO #TARGET
0430    MOVE #FOURTH TO #TARGET
0440    MOVE #FIFTH TO #TARGET
0450    MOVE #FIRST TO #TARGET
0460    MOVE #SECOND TO #TARGET
0470    MOVE #THIRD TO #TARGET
0480    MOVE #FOURTH TO #TARGET
0490    MOVE #FIFTH TO #TARGET
0500    END-IF
0510 END-FOR
0520 WRITE 5T 'NON ARRAY TIME' *TIMD (SETC.)
0530 *
0540 SETA. SETTIME
0550 FOR #LOOP = 1 TO 150000
0560 IF #CRITERIA = ' '
0570   IGNORE
0580   END-IF
0590 END-FOR
0600 WRITE 5T 'DUMMY FOR LOOP TIME' *TIMD (SETA.)
0610
0620 END
```

And the rather significant times:

```
PAGE #   1                  DATE:    Nov 30, 2001
PROGRAM: ARRAY08            LIBRARY: INSIDE

SUBSTRING TIME       43
NON ARRAY TIME       22
DUMMY FOR LOOP TIME       12
```

I remembered to include the "dummy" FOR loop, which is "overhead" for both loops. A true performance comparison, therefore, would be 43 - 12 versus 22 - 12; or 31 versus 10. That's a factor of three. Again, clearly, the REDEFINE'd simple variable far outperforms its alternatives. In this case, it is the SUBSTRING that suffers by comparison.

Here is a comparison of all three techniques.

```
* THIS PROGRAM SHOWS A COMPARISON OF ALL THE ARRAY
* REFERENCING TECHNIQUES
*
DEFINE DATA LOCAL
1 #ARRAY (A5/1:5)
1 REDEFINE #ARRAY
  2 #FIRST (A5)
  2 #SECOND (A5)
  2 #THIRD (A5)
  2 #FOURTH (A5)
  2 #FIFTH (A5)
1 #STRING (A25)
1 #INDEX (I4)
1 #TARGET (A5)
1 #CRITERIA (A3)
1 #LOOP (P7)
END-DEFINE
*
INCLUDE AATITLER
INCLUDE AASETC
*
SETB. SETTIME
FOR #LOOP = 1 TO 150000
IF #CRITERIA = ' '
   MOVE #ARRAY (1) TO #TARGET
   MOVE #ARRAY (2) TO #TARGET
   MOVE #ARRAY (3) TO #TARGET
   MOVE #ARRAY (4) TO #TARGET
   MOVE #ARRAY (5) TO #TARGET
   MOVE #ARRAY (1) TO #TARGET
   MOVE #ARRAY (2) TO #TARGET
   MOVE #ARRAY (3) TO #TARGET
   MOVE #ARRAY (4) TO #TARGET
   MOVE #ARRAY (5) TO #TARGET
   END-IF
END-FOR
WRITE 5T 'FIXED SUBSCRIPT TIME' *TIMD (SETB.)
*
SETC. SETTIME
FOR #LOOP = 1 TO 150000
IF #CRITERIA = ' '
   MOVE #FIRST TO #TARGET
   MOVE #SECOND TO #TARGET
   MOVE #THIRD TO #TARGET
   MOVE #FOURTH TO #TARGET
   MOVE #FIFTH TO #TARGET
   MOVE #FIRST TO #TARGET
   MOVE #SECOND TO #TARGET
   MOVE #THIRD TO #TARGET
   MOVE #FOURTH TO #TARGET
   MOVE #FIFTH TO #TARGET
   END-IF
END-FOR
WRITE 5T 'NON ARRAY TIME' *TIMD (SETC.)
*
SETA. SETTIME
FOR #LOOP = 1 TO 150000
IF #CRITERIA = ' '
   MOVE SUBSTRING (#STRING,1,5) TO #TARGET
   MOVE SUBSTRING (#STRING,6,5) TO #TARGET
   MOVE SUBSTRING (#STRING,11,5) TO #TARGET
   MOVE SUBSTRING (#STRING,16,5) TO #TARGET
   MOVE SUBSTRING (#STRING,21,5) TO #TARGET
   MOVE SUBSTRING (#STRING,1,5) TO #TARGET
   MOVE SUBSTRING (#STRING,6,5) TO #TARGET
   MOVE SUBSTRING (#STRING,11,5) TO #TARGET
   MOVE SUBSTRING (#STRING,16,5) TO #TARGET
   MOVE SUBSTRING (#STRING,21,5) TO #TARGET
   END-IF
END-FOR
WRITE 5T 'SUBSTRING TIME' *TIMD (SETA.)
*
SETD. SETTIME
FOR #LOOP = 1 TO 150000
IF #CRITERIA = ' '
  IGNORE
  END-IF
END-FOR
WRITE 5T 'DUMMY FOR LOOP TIME' *TIMD (SETD.)
END
```

And our output:

```
PAGE #   1                   DATE:    Nov 30, 2001
PROGRAM: ARRAY09             LIBRARY: INSIDE

FIXED SUBSCRIPT TIME       43
NON ARRAY TIME       22
SUBSTRING TIME       43
DUMMY FOR LOOP TIME       12
```

What do these numbers mean?

Actually, lets start out with what these numbers do NOT mean. They do not mean you should not use arrays and they do not mean that you should not use SUBSTRING.

They DO mean that when either a REDEFINEd simple variable or an array reference will do, use the simple variable. Similarly, when the same simple variable can be used in lieu of a SUBSTRING reference, use the simple variable.

However, there are numerous scenarios where arrays and SUBSTRING are obviously appropriate. Suppose I want, in turn, to test all the members of an array and perform actions based on their values. Clearly I do not want to "straight line code" this for the sake of saving a few machine cycles. Besides, depending on the size of the array, straight line code referencing #ONE, #TWO,...#NINETY-THREE etc. will almost certainly exceed Natural program size.

Note that SUBSTRING and array reference are identical. This is rather interesting, but not totally surprising. Why? Note that I have fixed starting positions and fixed sizes. This makes the comparison "fair". But suppose I wrote some "bad code". The following program is rather interesting. It contrasts bad array code, good array code, bad SUBSTRING code and good SUBSTRING code.

```
* THIS PROGRAM COMPARES SUBSTRING WITH ARRAYS;
* GOOD CODE WITH BAD
*
DEFINE DATA LOCAL
1 #STRING (A25)
1 REDEFINE #STRING
  2 #ARRAY (A5/1:5)
1 #INDEX (I4)
1 #TARGET (A5)
1 #CRITERIA (A3)
1 #LOOP (P7)
1 #ELEVEN (I4) INIT <11>
1 #FIVE (I4) INIT <5>
1 #THREE (I4) INIT <3>
END-DEFINE
*
INCLUDE AATITLER
INCLUDE AASETC
*
SETB. SETTIME
FOR #LOOP = 1 TO 500000
   MOVE SUBSTRING (#STRING,11,5) TO #TARGET
END-FOR
WRITE 5T 'GOOD SUBSTRING TIME' *TIMD (SETB.)
*
SETC. SETTIME
FOR #LOOP = 1 TO 500000
   MOVE SUBSTRING (#STRING,#ELEVEN,#FIVE) TO #TARGET
END-FOR
WRITE 5T 'BAD SUBSTRING TIME' *TIMD (SETC.)
*
SETD. SETTIME
FOR #LOOP = 1 TO 500000
   MOVE #ARRAY (3) TO #TARGET
END-FOR
WRITE 5T 'GOOD ARRAY TIME' *TIMD (SETD.)
*
*
SETA. SETTIME
FOR #LOOP = 1 TO 500000
   MOVE #ARRAY (#THREE) TO #TARGET
END-FOR
WRITE 5T 'BAD ARRAY TIME' *TIMD (SETA.)

END
```

There will be two sets of output. The first is from Natural 4 on my PC.

```
PAGE #   1                    DATE:    Dec 14, 2001
PROGRAM: ARRAY15              LIBRARY: INSIDE

GOOD SUBSTRING TIME       39
BAD SUBSTRING TIME       39
GOOD ARRAY TIME       39
BAD ARRAY TIME       39
```

The second output times are from a mainframe. One other difference, all loops were for 300,000 iterations rather than the 500,000 from the PC.

```
PAGE #   1                    DATE:    Dec 14, 2001
PROGRAM: ARRAY15              LIBRARY: INSIDE

GOOD SUBSTRING TIME       21
BAD SUBSTRING TIME       21
GOOD ARRAY TIME       10
BAD ARRAY TIME       10
```

Some observations:

Note that in all cases, the good equals the bad. What does this mean? Consider #ARRAY(3) and #ARRAY (#THREE). The times for these are the same. The only way they can be the same is if Natural is not taking advantage of “knowing” at compile time that #ARRAY (3) is a specific string of bytes. It appears that Natural is basically replacing the three with a pointer to a constant three; much as it would if you used a three in a COMPUTE statement. That is why a redefined variable is faster than the reference to a three as a subscript.

Taking into account the difference between the number of iterations, the mainframe SUBSTRING time, per 100,000 iterations, was .70 seconds while the PC time was .78 seconds. Again, rather interesting to note how close these times were, especially since my PC is an older 600MhZ system.

What is truly interesting is the comparison of the array time versus the substring time. On the PC, these times are the same. On the mainframe, the array times are half the substring times. I have an inquiry in to the development team as to why this is true. While I can see there should be some difference, a doubling seems excessive. Consider MOVE #ARRAY(#INDEX) versus MOVE SUBSTRING (#STRING,#START,#NUMBER).

For the array, Natural has to take the start address for the array, add (#INDEX - 1) times the length of #ARRAY to find the start of the appropriate entry, then move the appropriate number of characters (length of #ARRAY). For SUBSTRING, there are similar tasks. Natural must take the start address for the array, add (#START - 1) to get the starting position (less work than for the array), then move the number of characters specified in the third argument. It just doesn’t seem that the difference should be that great. Note that the PC times are identical, which is closer to what I would expect.

Summary

Actually, the last section of this article covered the main points:

If you need a specific occurrence of an array, use REDEFINE to create a simple variable for reference as opposed to any array or SUBSTRING reference.

On a mainframe, or on the PC, there is no difference between a constant subscript, say #ARRAY (3) and a variable subscript, say #ARRAY (#THREE), where #THREE is an I4 variable with value three. HOWEVER, neither is as good as a simple variable that has been REDEFINEd.

On a mainframe, array operations outperform SUBSTRING operations by a considerable margin. On a PC, the times are identical. ❖

More on Arrays

About midway through Thomas's presentation he had a performance hint that I wanted to take exception to. Not that there was anything wrong with his statement, there wasn't. However, there was a lot of discussion that should accompany this particular warning. So, here is that discussion.

Thomas's comment had to do with the following two definitions:

DEFINE DATA LOCAL	DEFINE DATA LOCAL
1 #GROUP (100)	1 #A (A10/1:100)
2 #A (A10)	1 #B (N10/1:100)
2 #B (N10)	END-DEFINE
END-DEFINE	

As will be shown below, there is a substantial performance difference between seemingly simple commands like RESET #A (*) #B (*).

First, however, lets talk a bit about the difference between the two structures. The major difference is simply sequence in memory. The "group" structure appears in memory as:

#A (1) #B (1) #A (2) #B (2) ..... #A (100) #B (100)

In other words, the data is stored "group-wise". By contrast, the "fields" structure is:

#A (1) #A (2) #A (3)......#A (100) #B (1) #B (2) #B (3)....#B (100)

In other words, the data is stored "field-wise".

Thomas's point was that some commands, like RESET, work more efficiently when the fields being RESET are contiguous. Here is an example that contrasts the time required for RESET's using both structures.

```
* THIS PROGRAM CONTRASTS TWO WAYS TO DEFINE AN
* ARRAY AND THE RESULTANT COST OF SIMPLE RESETS
*
DEFINE DATA LOCAL
1 #LOOP (P7)
1 #GROUP (100)
  2 #A (A10)
  2 #B (A5)
1 #AA (A10/1:100)
1 #BB (A5/1:100)
END-DEFINE
*
INCLUDE AATITLER
INCLUDE AASETC
*
SETA. SETTIME
FOR #LOOP = 1 TO 100000
RESET #A (*) #B (*)
END-FOR
WRITE 5T 'GROUP ARRAY TIME' *TIMD (SETA.) //
*
SETB. SETTIME
FOR #LOOP = 1 TO 100000
RESET #AA (*) #BB (*)
END-FOR
WRITE 5T 'SEPARATE ARRAY TIME' *TIMD (SETB.) //
*
SETC. SETTIME
FOR #LOOP = 1 TO 100000
IGNORE
END-FOR
WRITE 5T 'FOR LOOP TIME' *TIMD (SETC.)
*
END
```

And here is our output:

```
PAGE #   1                  DATE:    Nov 30, 2001
PROGRAM: ARRAY10            LIBRARY: INSIDE

GROUP ARRAY TIME      108


SEPARATE ARRAY TIME       14


FOR LOOP TIME       6
```

Take a look at the times!! Subtracting the common FOR loop overhead, the times are 102 and 8. That's a factor of almost thirteen. So, is this a reason for never using the "group-wise" structure? Absolutely not!! Is it a reason for sometimes not using the "group-wise" structure? Maybe.

First, as will be shown below, not all statements display the same performance differential. Second, there are applications where all processing is "group-wise" rather than "field-wise". For such applications, the group-wise structure is clearly most appropriate.

Lets start by looking at several statements and their relative efficiency given the two different structures. First, IF statements.

```
0010 * THIS PROGRAM CONTRASTS TWO WAYS TO DEFINE AN
0020 * ARRAY AND THE RESULTANT COST OF IF STATEMENTS
0030 *
0040 DEFINE DATA LOCAL
0050 1 #LOOP (P7)
0060 1 #GROUP (100)
0070   2 #A (A10)
0080   2 #B (A5)
0090 1 #AA (A10/1:100)
0100 1 #BB (A5/1:100)
0110 END-DEFINE
0120 *
0130 INCLUDE AATITLER
0140 INCLUDE AASETC
0150 *
0160 SETA. SETTIME
0170 FOR #LOOP = 1 TO 100000
0180 IF #A (*) = 'X'
0190   IGNORE
0200   END-IF
0210 END-FOR
0220 WRITE 5T 'GROUP ARRAY TIME' *TIMD (SETA.) //
0230 *
0240 SETB. SETTIME
0250 FOR #LOOP = 1 TO 100000
0260 IF #AA (*) = 'X'
0270   IGNORE
0280   END-IF
0290 END-FOR
0300 WRITE 5T 'SEPARATE ARRAY TIME' *TIMD (SETB.) //
0310 *
```

```
0320 SETC. SETTIME
0330 FOR #LOOP = 1 TO 100000
0340 IGNORE
0350 END-FOR
0360 WRITE 5T 'FOR LOOP TIME' *TIMD (SETC.)
0370 *
0380 END
```

```
PAGE #   1                  DATE:    Nov 30, 2001
PROGRAM: ARRAY11            LIBRARY: INSIDE

GROUP ARRAY TIME      163


SEPARATE ARRAY TIME      162


FOR LOOP TIME       6
```

In fairly recent articles we have discussed how much more efficient it is to IF #ARRAY(*) rather than testing #ARRAY (#LOOP) inside of a FOR loop on #LOOP. The question, answered above, is whether there is any difference between the two type of structures in terms of IF (*) efficiency. The answer is clearly no. The IF deals with each occurrence individually, hence their location is irrelevant.

In case you missed the article which demonstrates why you should not set up a FOR loop to test all the occurrences of an array for a given value, here is a program that offers rather graphic evidence why you should not do this.

```
0010 * THIS PROGRAM CONTRASTS TWO WAYS TO DEFINE AN
0020 * ARRAY AND THE RESULTANT COST OF IF STATEMENTS
0030 *
0040 DEFINE DATA LOCAL
0050 1 #LOOP (P7)
0060 1 #LOOP2 (P7)
0070 1 #GROUP (10)
0080   2 #A (A10)
0090   2 #B (A5)
0100 1 #AA (A10/1:10)
0110 1 #BB (A5/1:10)
0120 END-DEFINE
0130 *
0140 INCLUDE AATITLER
0150 INCLUDE AASETC
0160 *
0170 SETA. SETTIME
0180 FOR #LOOP = 1 TO 100000
0190 IF #A (*) = 'X'
0200   IGNORE
0210   END-IF
0220 END-FOR
0230 WRITE 5T 'GROUP ARRAY TIME' *TIMD (SETA.) //
```

```
0240 *
0250 SETB. SETTIME
0260 FOR #LOOP = 1 TO 100000
0270 IF #AA (*) = 'X'
0280   IGNORE
0290   END-IF
0300 END-FOR
0310 WRITE 5T 'SEPARATE ARRAY TIME' *TIMD (SETB.) //
0320 *
0330 SETC. SETTIME
0340 FOR #LOOP = 1 TO 100000
0350 IGNORE
0360 END-FOR
0370 WRITE 5T 'DUMMY FOR LOOP TIME' *TIMD (SETC.) //
0380 *
0390 SETD. SETTIME
0400 FOR #LOOP = 1 TO 100000
0410 IF #A (1) = 'X'
0420 OR #A (2) = 'X'
0430 OR #A (3) = 'X'
0440 OR #A (4) = 'X'
0450 OR #A (5) = 'X'
0460 OR #A (6) = 'X'
0470 OR #A (7) = 'X'
0480 OR #A (8) = 'X'
0490 OR #A (9) = 'X'
0500 OR #A (10) = 'X'
0510 IGNORE
0520 END-IF
0530 END-FOR
0540 WRITE 5T 'SEPARATE OR TIME' *TIMD (SETD.) //
0550 *
0560 *
0570 SETG. SETTIME
0580 FOR #LOOP = 1 TO 100000
0590   FOR #LOOP2 = 1 TO 10
0600   IF #A (#LOOP2) = 'X'
0610      IGNORE
0620      END-IF
0630      END-FOR
0640 END-FOR
0650 WRITE 5T 'FOR LOOP TIME' *TIMD (SETG.)
0660
0670 END
```

And the output:

```
PAGE #   1                  DATE:    Dec 14, 2001
PROGRAM: ARRAY12            LIBRARY: INSIDE

GROUP ARRAY TIME       25


SEPARATE ARRAY TIME       26


DUMMY FOR LOOP TIME        5


SEPARATE OR TIME       38


FOR LOOP TIME      105
```

Again, note that the IF(*) notation has the same time regardless of whether a group or field approach has been employed for the array. The IF against individual occurrences is 50% more than the IF(*) notation, and the FOR loop.... Well, you can see how bad it is. I find the FOR loop everywhere. It is horribly inefficient. Do not use it. Scan existing programs and get rid of it. Tell other people about it. Okay, I'll stop now.

MOVE statements

As we saw above, there was quite a difference between RESETing an array that is part of a group and an array that is a level one variable. I wondered about a simple MOVE statement.

```
0010 * THIS PROGRAM CONTRASTS TWO WAYS TO DEFINE AN
0020 * ARRAY AND THE RESULTANT COST OF SIMPLE MOVE'S
0030 *
0040 DEFINE DATA LOCAL
0050 1 #LOOP (P7)
0060 1 #GROUP (100)
0070   2 #A (A10)
0080   2 #B (A5)
0090 1 #AA (A10/1:100)
0100 1 #BB (A5/1:100)
0110 END-DEFINE
0120 *
0130 INCLUDE AATITLER
0140 INCLUDE AASETC
0150 *
0160 SETA. SETTIME
0170 FOR #LOOP = 1 TO 100000
0180 MOVE 'ABCDE' TO #B (*)
0190 END-FOR
0200 WRITE 5T 'GROUP ARRAY TIME' *TIMD (SETA.) //
0210 *
0220 SETB. SETTIME
0230 FOR #LOOP = 1 TO 100000
0240 MOVE 'ABCDE' TO #BB (*)
0250 END-FOR
0260 WRITE 5T 'SEPARATE ARRAY TIME' *TIMD (SETB.) //
0270 *
0280 SETC. SETTIME
0290 FOR #LOOP = 1 TO 100000
0300 IGNORE
0310 END-FOR
0320 WRITE 5T 'FOR LOOP TIME' *TIMD (SETC.)
0330 *
0340 END
```

And our output:

```
PAGE #   1                  DATE:    Dec 15, 2001
PROGRAM: ARRAY17            LIBRARY: INSIDE

GROUP ARRAY TIME       66


SEPARATE ARRAY TIME       67


FOR LOOP TIME        5
```

Just like the IF statement comparison, no difference between the group array structure and the field array structure. Indeed, the only commands I found a difference in performance were the RESET and the commands that are the subject of the next article on READ and WRITE WORK FILE. ❖

Work Files

Actually, this will be two articles in one. The first will address WRITE WORK FILE, the second READ WORK FILE. Interspersed, there will be what I expect will be an interesting and enlightening tangent.

WRITE WORK FILE

Lets start with what might be considered the less interesting of the two work file commands, namely WRITE WORK FILE (WWF from here on; I get tired of typing so many letters).

As I mentioned, Thomas was a bit short of time. There was a brief note towards the end of his handout (actually, the next to last item) which said, “When many fields are specified in a READ/ WRITE WORK FILE statement, the data transfer works faster if the fields specified are defined in a contiguous manner”.

Now this is one of those things that I heard in the early days of Natural and never mentally challenged. It seemed to make sense. As mentioned earlier, there I was on a plane from Frankfurt back home to Philadelphia after the ENPUG conference. I had time available since I had packed the book I was reading in the suitcase I checked rather than my laptop bag. I started wondering about WWF. Is this a compile time check for contiguity, or a run time check. My guess was that it had to be compile time. A run time check would be too expensive. I keyed in a simple starting program. The results were, to say the least, interesting.

```
0010 DEFINE DATA LOCAL
0020 1 #ARRAY (A3/1:50) INIT ALL <'ABC'>
0030 1 #LOOP (P7)
0040 1 #STRING (A200)
0050 END-DEFINE
0060 *
0070 INCLUDE AATITLER
0080 INCLUDE AASETC
0090 *
0100 ST2. SETTIME
0110 FOR #LOOP = 1 TO 250000
0120 WRITE WORK FILE 2
0130 #ARRAY (1) #ARRAY (2) #ARRAY (3) #ARRAY (4) #ARRAY (5)
0140 END-FOR
0150 WRITE 10T 'ORDERED TIME' *TIMD (ST2.)
```

```
0160 *
0170 ST1. SETTIME
0180 FOR #LOOP = 1 TO 250000
0190 WRITE WORK FILE 1
0200 #ARRAY (1) #ARRAY (2) #ARRAY (3) #ARRAY (5) #ARRAY (4)
0210 END-FOR
0220"WRITE 10T 'NON ORDERED TIME' *TIMD (ST1.) //
0230
0240 END
```

And the somewhat unexpected output.

```
     PAGE #   1                  DATE:    Nov 22, 2001
     PROGRAM: WORK14             LIBRARY: INSIDE

          ORDERED TIME      40
          NON ORDERED TIME      39
```

Well, that was interesting. Given that the difference is quite minuscule, suppose that in reality they are equal. That still says that ordering the variables in the WWF statement did not improve performance. I thought about this for awhile. Maybe the effect of ordering only shows up when you have lots of values. I tried a larger array.

```
0010 DEFINE DATA LOCAL
0020 1 #ARRAY (A3/1:50) INIT ALL <'ABC'>
0030 1 #LOOP (P7)
0040 1 #STRING (A200)
0050 END-DEFINE
0060 *
0070 INCLUDE AATITLER
0080 INCLUDE AASETC
0090 *
0100 ST2. SETTIME
0110 FOR #LOOP = 1 TO 250000
0120 WRITE WORK FILE 2
0130 #ARRAY (1) #ARRAY (2) #ARRAY (3) #ARRAY (4) #ARRAY (5)
0140 #ARRAY(46) #ARRAY(47) #ARRAY(48) #ARRAY(49)#ARRAY (50)
0150 #ARRAY (6) #ARRAY (7) #ARRAY (8) #ARRAY(9) #ARRAY (10)
0160 #ARRAY(41) #ARRAY (42) #ARRAY(43)#ARRAY(44)#ARRAY (45)
0170 #ARRAY(11) #ARRAY(12)#ARRAY (13)#ARRAY(14)#ARRAY (15)
0180 #ARRAY(36) #ARRAY(37)#ARRAY (38)#ARRAY(39)#ARRAY (40)
0190 #ARRAY(16) #ARRAY(17)#ARRAY (18)#ARRAY(19)#ARRAY (20)
0200 #ARRAY(31) #ARRAY(32) #ARRAY(33) #ARRAY(34) #ARRAY(35)
0210 #ARRAY(26) #ARRAY(27) #ARRAY(28) #ARRAY(29) #ARRAY(30)
0220 #ARRAY(21) #ARRAY(22) #ARRAY(23) #ARRAY(24) #ARRAY(25)
0230 END-FOR
0240 WRITE 10T 'NON ORDERED TIME' *TIMD (ST2.)
0250 *
0260 ST1. SETTIME
0270 FOR #LOOP = 1 TO 250000
0280 WRITE WORK FILE 1
0290 #ARRAY(1) #ARRAY(2) #ARRAY(3) #ARRAY (4)   #ARRAY (5)
0300 #ARRAY(6) #ARRAY(7) #ARRAY(8) #ARRAY (9)   #ARRAY (10)
0310 #ARRAY(11) #ARRAY(12) #ARRAY(13) #ARRAY(14) #ARRAY(15)
0320 #ARRAY(16) #ARRAY(17) #ARRAY(18) #ARRAY(19) #ARRAY(20)
0330 #ARRAY(21) #ARRAY(22) #ARRAY(23) #ARRAY(24) #ARRAY(25)
0340 #ARRAY(26) #ARRAY(27) #ARRAY(28) #ARRAY(29) #ARRAY(30)
0350 #ARRAY(31) #ARRAY(32) #ARRAY(33) #ARRAY(34) #ARRAY(35)
0360 #ARRAY(36) #ARRAY(37) #ARRAY(38) #ARRAY(39) #ARRAY(40)
0370 #ARRAY(41) #ARRAY(42) #ARRAY(43) #ARRAY(44) #ARRAY(45)
0380 #ARRAY(46) #ARRAY(47) #ARRAY(48) #ARRAY(49) #ARRAY(50)
0390 END-FOR
0400 WRITE 10T 'ORDERED TIME' *TIMD (ST1.) //
0410
0420 END
```

And the output I expected, but wished I did not see.

```
PAGE #   1                   DATE:    Nov 22, 2001
PROGRAM: WORK13              LIBRARY: INSIDE

     NON ORDERED TIME      259
     ORDERED TIME      260
```

So much for that theory. The two are basically identical. I decided to add to the previous example with something I was certain would be more efficient.

```
0010 DEFINE DATA LOCAL
0020 1 #ARRAY (A3/1:50) INIT ALL <'ABC'>
0030 1 #LOOP (P7)
0040 1 #STRING (A200)
0050 END-DEFINE
0060 *
0070 INCLUDE AATITLER
0080 INCLUDE AASETC
0090 *
0100 ST2. SETTIME
0110 FOR #LOOP = 1 TO 250000
0120 WRITE WORK FILE 2
0130 #ARRAY (1) #ARRAY (2) #ARRAY (3) #ARRAY (4) #ARRAY (5)
0140 #ARRAY(46) #ARRAY(47) #ARRAY(48) #ARRAY(49) #ARRAY(50)
0150 #ARRAY (6) #ARRAY (7) #ARRAY (8) #ARRAY(9)  #ARRAY(10)
0160 #ARRAY(41) #ARRAY (42)#ARRAY(43) #ARRAY(44) #ARRAY(45)
0170 #ARRAY(11) #ARRAY(12) #ARRAY(13) #ARRAY(14) #ARRAY(15)
0180 #ARRAY(36) #ARRAY(37) #ARRAY(38) #ARRAY(39) #ARRAY(40)
0190 #ARRAY(16) #ARRAY(17) #ARRAY(18) #ARRAY(19) #ARRAY(20)
0200 #ARRAY(31) #ARRAY(32) #ARRAY(33) #ARRAY(34) #ARRAY(35)
0210 #ARRAY(26) #ARRAY(27) #ARRAY(28) #ARRAY(29) #ARRAY(30)
0220 #ARRAY(21) #ARRAY(22) #ARRAY(23) #ARRAY(24) #ARRAY(25)
0230 END-FOR
0240 WRITE 10T 'NON ORDERED TIME' *TIMD (ST2.) //
0250 *
0260 ST1. SETTIME
0270 FOR #LOOP = 1 TO 250000
0280 WRITE WORK FILE 1
0290 #ARRAY(1) #ARRAY(2) #ARRAY(3) #ARRAY (4)   #ARRAY (5)
0300 #ARRAY(6) #ARRAY(7) #ARRAY(8) #ARRAY (9)   #ARRAY (10)
0310 #ARRAY(11) #ARRAY(12) #ARRAY(13) #ARRAY(14) #ARRAY(15)
0320 #ARRAY(16) #ARRAY(17) #ARRAY(18) #ARRAY(19) #ARRAY(20)
0330 #ARRAY(21) #ARRAY(22) #ARRAY(23) #ARRAY(24) #ARRAY(25)
0340 #ARRAY(26) #ARRAY(27) #ARRAY(28) #ARRAY(29) #ARRAY(30)
0350 #ARRAY(31) #ARRAY(32) #ARRAY(33) #ARRAY(34) #ARRAY(35)
0360 #ARRAY(36) #ARRAY(37) #ARRAY(38) #ARRAY(39) #ARRAY(40)
0370 #ARRAY(41) #ARRAY(42) #ARRAY(43) #ARRAY(44) #ARRAY(45)
0380 #ARRAY(46) #ARRAY(47) #ARRAY(48) #ARRAY(49) #ARRAY(50)
0390 END-FOR
0400 WRITE 10T 'ORDERED TIME' *TIMD (ST1.) //
0410 *
0420 ST3. SETTIME
0430 FOR #LOOP = 1 TO 250000
0440 WRITE WORK FILE 1
0450 #ARRAY (*)
0460 END-FOR
0470 WRITE 10T 'ORDERED ARRAY TIME' *TIMD (ST3.) //
0480
0490 END
```

The output was consistent with my expectations:

```
PAGE #   1                   DATE:    Nov 25, 2001
PROGRAM: WORK16              LIBRARY: INSIDE

     NON ORDERED TIME      255


     ORDERED TIME      253


     ORDERED ARRAY TIME      92
```

Aha. I was finally right about something. The use of “array syntax” reduced the time by about two thirds. Quite a savings.

TANGENT

About this time I was really curious what was going on. Thusfar, I had run all my examples on my PC using Natural 4. I decided to run the previous program on a mainframe. Since timings on a mainframe tend to vary quite a bit (based on what else is running) I ran the program several times. Here are the averages.

```
PAGE #   1                   DATE:    Nov 25, 2001
PROGRAM: WORK16              LIBRARY: INSIDE

     NON ORDERED TIME      351


     ORDERED TIME      335


     ORDERED ARRAY TIME      186
```

Now if you are just looking at the numbers in the box above, you have noticed that the ordered time is a bit better relative to the non ordered time. However, given the variation in times due to what else was on the computer, my guess is that the “non ordered” and “ordered” times are in reality equal, as they are on the PC. The ordered array time is fastest, as was the case on the PC.

Now look at the PC times (above) versus the mainframe times. The PC times are considerably FASTER than the mainframe times. Now some explanation is relevant. The PC times are from my 600 MhZ PC, which is running just my program. The mainframe times are from an evening batch run competing with all the other batch jobs for machine cycles. Even so, the mainframe times are 30-50 percent higher than the PC times. As mentioned in the last issue; I will be investigating this further. This is really an eye opener. Many "mainframers" have adapted the viewpoint that "real work" cannot be done on PC's. The numbers I have seen thusfar would seem to indicate this is not so.

Back to the main thread

I was curious about the use of a Group name here. So I first set up a group name that "equated to" an array. The question is whether Natural would "expand" the Group Name as #ARRAY (*) or #ARRAY (1) #ARRAY (2)... etc.

Since the times for these two expansions were so different, we should be able to tell which expansion took place.

```
DEFINE DATA LOCAL
1 GROUP-NAME
  2 #ARRAY (A3/1:50) INIT ALL <'ABC'>
1 #LOOP (P7)
1 #STRING (A200)
END-DEFINE
*
INCLUDE AATITLER
INCLUDE AASETC
*
ST2. SETTIME
FOR #LOOP = 1 TO 250000
WRITE WORK FILE 2
GROUP-NAME
END-FOR
WRITE 10T 'GROUP TIME' *TIMD (ST2.) //
*
ST1. SETTIME
FOR #LOOP = 1 TO 250000
WRITE WORK FILE 1
#ARRAY (1) #ARRAY(2) #ARRAY(3) #ARRAY(4) #ARRAY (5)
#ARRAY (6) #ARRAY(7) #ARRAY(8) #ARRAY(9) #ARRAY (10)
#ARRAY(11) #ARRAY(12) #ARRAY(13) #ARRAY(14) #ARRAY(15)
#ARRAY(16) #ARRAY(17) #ARRAY(18) #ARRAY(19) #ARRAY(20)
#ARRAY(21) #ARRAY(22) #ARRAY(23) #ARRAY(24) #ARRAY(25)
#ARRAY(26) #ARRAY(27) #ARRAY(28) #ARRAY(29) #ARRAY(30)
#ARRAY(31) #ARRAY(32) #ARRAY(33) #ARRAY(34) #ARRAY(35)
#ARRAY(36) #ARRAY(37) #ARRAY(38) #ARRAY(39) #ARRAY(40)
#ARRAY(41) #ARRAY(42) #ARRAY(43) #ARRAY(44) #ARRAY 45)
#ARRAY(46) #ARRAY(47) #ARRAY(48) #ARRAY(49) #ARRAY(50)
END-FOR
WRITE 10T 'ORDERED TIME' *TIMD (ST1.) //
*
ST3. SETTIME
FOR #LOOP = 1 TO 250000
WRITE WORK FILE 1
#ARRAY (*)
END-FOR
WRITE 10T 'ORDERED ARRAY TIME' *TIMD (ST3.) //

END
```

And the output, which answers our question.

```
PAGE #   1                    DATE:    Dec 02, 2001
PROGRAM: WORK17               LIBRARY: INSIDE

     GROUP TIME       94


     ORDERED TIME      254


     ORDERED ARRAY TIME       91
```

Clearly "GROUP-NAME" was expanded as #ARRAY (*) since the time for this was very close to the last time, not the individual occurrences time.

Okay, suppose the Group contained separate fields, not an array. Would there be any benefit from having contiguous fields?

```
DEFINE DATA LOCAL
1 GROUP-NAME
  2 #A1 (A3) INIT  <'ABC'>
  2 #A2 (A3) INIT  <'ABC'>
  2 #A3 (A3) INIT  <'ABC'>
  2 #A4 (A3) INIT  <'ABC'>
  2 #A5 (A3) INIT  <'ABC'>
  2 #A6 (A3) INIT  <'ABC'>
  2 #A7 (A3) INIT  <'ABC'>
  2 #A8 (A3) INIT  <'ABC'>
  2 #A9 (A3) INIT  <'ABC'>
  2 #A10 (A3) INIT  <'ABC'>
  2 #A11 (A3) INIT  <'ABC'>
  2 #A12 (A3) INIT  <'ABC'>
  2 #A13 (A3) INIT  <'ABC'>
  2 #A14 (A3) INIT  <'ABC'>
  2 #A15 (A3) INIT  <'ABC'>
  2 #A16 (A3) INIT  <'ABC'>
  2 #A17 (A3) INIT  <'ABC'>
  2 #A18 (A3) INIT  <'ABC'>
  2 #A19 (A3) INIT  <'ABC'>
  2 #A20 (A3) INIT  <'ABC'>
  2 #A21 (A3) INIT  <'ABC'>
  2 #A22 (A3) INIT  <'ABC'>
  2 #A23 (A3) INIT  <'ABC'>
  2 #A24 (A3) INIT  <'ABC'>
  2 #A25 (A3) INIT  <'ABC'>
1 #LOOP (P7)
1 #STRING (A200)
END-DEFINE
*
INCLUDE AATITLER
INCLUDE AASETC
```

```
*
ST2. SETTIME
FOR #LOOP = 1 TO 250000
WRITE WORK FILE 2
GROUP-NAME
END-FOR
WRITE 10T 'GROUP TIME' *TIMD (ST2.) //
*
ST1. SETTIME
FOR #LOOP = 1 TO 250000
WRITE WORK FILE 1
#A1 #A2 #A3 #A4 #A5 #A6 #A7 #A8 #A9 #A10
#A11 #A12 #A13 #A14 #A15 #A16 #A17 #A18 #A19 #A20
#A21 #A22 #A23 #A24 #A25
END-FOR
WRITE 10T 'ORDERED TIME' *TIMD (ST1.) //
*
ST3. SETTIME
FOR #LOOP = 1 TO 250000
WRITE WORK FILE 1
#A1 #A25 #A3 #A4 #A5 #A14 #A7 #A21 #A9 #A10
#A11 #A12 #A13 #A6 #A15 #A16 #A17 #A18 #A19 #A20
#A8 #A22 #A23 #A24 #A2
END-FOR
WRITE 10T 'UNORDERED TIME' *TIMD (ST3.) //

END
```

And our output.

```
PAGE #   1                  DATE:    Dec 02, 2001
PROGRAM: WORK18             LIBRARY: INSIDE

     GROUP TIME       48


     ORDERED TIME       48


     UNORDERED TIME       49
```

The answer is clearly not. The ordered and group times were the same. The unordered time was a tenth of a second behind, for 250,000 records. Basically, all the times are the same. Rather interesting. Ordering the fields for WRITE WORK FILE has no apparent effect on elapsed time. The only thing that does affect elapsed time, quite substantially, is “array notation”.

READ WORK FILE
and
READ WORK FILE RECORD

Okay, on to READ WORK FILE. First a note. At the end of this article we will consider READ WORK FILE RECORD. Why not consider it from the start? In a very real sense, RWF RECORD only works with single variables (okay, the single variable can be an array). Some of you are saying, “not true, I can use a Group Name with a RWF RECORD”. Yes, you can, but, as you will see, Natural is ignoring all the variable names within the group. So, be patient for awhile, we will look at RWF first.

We will start off with a comparison similar to one we did with WWF; namely, we will look at ordered fields, unordered fields, and a group name which equates to ordered fields.

```
* THIS PROGRAM CREATES A WORK FILE WITH 10000 RECORDS
*
* THEN WE COMPARE THE TIME TO READ THIS WORK FILE
* USING A GROUP NAME, SEPARATE, BUT ORDERED
* FIELD NAMES, AND UNORDERED FIELD NAMES
*
DEFINE DATA LOCAL
1 #GROUP
  2 #A1       (A5) INIT <'ABCDE'>
  2 #A2       (A5) INIT <'ABCDE'>
  2 #A3       (A5) INIT <'ABCDE'>
  2 #A4       (A5) INIT <'ABCDE'>
  2 #A5       (A5) INIT <'ABCDE'>
  2 #A6       (A5) INIT <'ABCDE'>
  2 #A7       (A5) INIT <'ABCDE'>
  2 #A8       (A5) INIT <'ABCDE'>
  2 #A9       (A5) INIT <'ABCDE'>
  2 #A10      (A5) INIT <'ABCDE'>
  2 #A11      (A5) INIT <'ABCDE'>
  2 #A12      (A5) INIT <'ABCDE'>
  2 #A13      (A5) INIT <'ABCDE'>
  2 #A14      (A5) INIT <'ABCDE'>
  2 #A15      (A5) INIT <'ABCDE'>
  2 #A16      (A5) INIT <'ABCDE'>
  2 #A17      (A5) INIT <'ABCDE'>
  2 #A18      (A5) INIT <'ABCDE'>
  2 #A19      (A5) INIT <'ABCDE'>
  2 #A20      (A5) INIT <'ABCDE'>
  2 #A21      (A5) INIT <'ABCDE'>
  2 #A22      (A5) INIT <'ABCDE'>
  2 #A23      (A5) INIT <'ABCDE'>
  2 #A24      (A5) INIT <'ABCDE'>
  2 #A25      (A5) INIT <'ABCDE'>
  2 #A26      (A5) INIT <'ABCDE'>
  2 #A27      (A5) INIT <'ABCDE'>
  2 #A28      (A5) INIT <'ABCDE'>
  2 #A29      (A5) INIT <'ABCDE'>
  2 #A30      (A5) INIT <'ABCDE'>
  2 #A31      (A5) INIT <'ABCDE'>
  2 #A32      (A5) INIT <'ABCDE'>
  2 #A33      (A5) INIT <'ABCDE'>
  2 #A34      (A5) INIT <'ABCDE'>
  2 #A35      (A5) INIT <'ABCDE'>
  2 #A36      (A5) INIT <'ABCDE'>
  2 #A37      (A5) INIT <'ABCDE'>
  2 #A38      (A5) INIT <'ABCDE'>
  2 #A39      (A5) INIT <'ABCDE'>
  2 #A40      (A5) INIT <'ABCDE'>
  2 #A41      (A5) INIT <'ABCDE'>
  2 #A42      (A5) INIT <'ABCDE'>
```

```
0510   2 #A43       (A5) INIT <'ABCDE'>
0520   2 #A44       (A5) INIT <'ABCDE'>
0530   2 #A45       (A5) INIT <'ABCDE'>
0540   2 #A46       (A5) INIT <'ABCDE'>
0550   2 #A47       (A5) INIT <'ABCDE'>
0560   2 #A48       (A5) INIT <'ABCDE'>
0570   2 #A49       (A5) INIT <'ABCDE'>
0580   2 #A50       (A5) INIT <'ABCDE'>
0590   1 #LOOP           (P7)
0600 END-DEFINE
0610 *
0620 INCLUDE AATITLER
0630 INCLUDE AASETC
0640 *
0650 FOR #LOOP = 1 TO 50000
0660 WRITE WORK FILE 1 #GROUP
0670 END-FOR
0680 *
0710 SETA. SETTIME
0720 READ WORK FILE  1  #GROUP
0730 END-WORK
0740 WRITE // 10T 'READ WORK FILE GROUP TIME' *TIMD (SETA.)
0750 *
0760 *
0770 SETB. SETTIME
0780 READ WORK FILE  1
0790 #A1 #A2 #A3 #A4 #A5 #A6 #A7 #A8 #A9 #A10
0800 #A11 #A12 #A13 #A14 #A15 #A16 #A17 #A18 #A19 #A20
0810 #A21 #A22 #A23 #A24 #A25 #A26 #A27 #A28 #A29 #A30
0820 #A31 #A32 #A33 #A34 #A35 #A36 #A37 #A38 #A39 #A40
0830 #A41 #A42 #A43 #A44 #A45 #A46 #A47 #A48 #A49 #A50
0840 END-WORK
0850 WRITE // 10T 'READ WORK FILE ORDERED FIELDS TIME'
0855             *TIMD (SETB.)
0860 *
0870 SETC. SETTIME
0880 READ WORK FILE  1
0890 #A50 #A2 #A3 #A4 #A5 #A6 #A7 #A8 #A49 #A10
0900 #A11 #A43 #A13 #A14 #A15 #A16 #A17 #A18 #A19 #A20
0910 #A21 #A22 #A23 #A24 #A25 #A26 #A27 #A28 #A29 #A30
0920 #A31 #A32 #A33 #A34 #A35 #A36 #A37 #A38 #A39 #A40
0930 #A41 #A42 #A12 #A44 #A45 #A46 #A47 #A48 #A9 #A1
0940 END-WORK
0950 WRITE // 10T 'READ WORK FILE NON ORDERED FIELDS TIME'
0955         *TIMD (SETC.)
0960 *
0990 END
```

And the not surprising times.

```
PAGE #   1                  DATE:    Dec 16, 2001
PROGRAM: READW06            LIBRARY: INSIDE


     READ WORK FILE GROUP TIME       18


     READ WORK FILE ORDERED FIELDS TIME       18


     READ WORK FILE NON ORDERED FIELDS TIME       17
```

Well, that pretty well says it all with regard to ordering for READ WORK FILE; it doesn't matter. Why doesn't ordering matter? More in a moment.

Now lets look at something that I thought would matter.

```
0010 * THIS PROGRAM CREATES A WORK FILE WITH 10000 RECORDS
0020 *
0030 * THEN WE COMPARE THE TIME TO READ THIS WORK FILE
0040 * USING A GROUP NAME, AND AN ARRAY
0060 *
0070 DEFINE DATA LOCAL
0080 1 #GROUP
0090   2 #A1        (A5) INIT <'ABCDE'>
0100   2 #A2        (A5) INIT <'ABCDE'>
0110   2 #A3        (A5) INIT <'ABCDE'>
0120   2 #A4        (A5) INIT <'ABCDE'>
0130   2 #A5        (A5) INIT <'ABCDE'>
0140   2 #A6        (A5) INIT <'ABCDE'>
0150   2 #A7        (A5) INIT <'ABCDE'>
0160   2 #A8        (A5) INIT <'ABCDE'>
0170   2 #A9        (A5) INIT <'ABCDE'>
0180   2 #A10       (A5) INIT <'ABCDE'>
0190   2 #A11       (A5) INIT <'ABCDE'>
0200   2 #A12       (A5) INIT <'ABCDE'>
0210   2 #A13       (A5) INIT <'ABCDE'>
0220   2 #A14       (A5) INIT <'ABCDE'>
0230   2 #A15       (A5) INIT <'ABCDE'>
0240   2 #A16       (A5) INIT <'ABCDE'>
0250   2 #A17       (A5) INIT <'ABCDE'>
0260   2 #A18       (A5) INIT <'ABCDE'>
0270   2 #A19       (A5) INIT <'ABCDE'>
0280   2 #A20       (A5) INIT <'ABCDE'>
0290   2 #A21       (A5) INIT <'ABCDE'>
0300   2 #A22       (A5) INIT <'ABCDE'>
0310   2 #A23       (A5) INIT <'ABCDE'>
0320   2 #A24       (A5) INIT <'ABCDE'>
0330   2 #A25       (A5) INIT <'ABCDE'>
0340   2 #A26       (A5) INIT <'ABCDE'>
0350   2 #A27       (A5) INIT <'ABCDE'>
0360   2 #A28       (A5) INIT <'ABCDE'>
0370   2 #A29       (A5) INIT <'ABCDE'>
0380   2 #A30       (A5) INIT <'ABCDE'>
0390   2 #A31       (A5) INIT <'ABCDE'>
0400   2 #A32       (A5) INIT <'ABCDE'>
0410   2 #A33       (A5) INIT <'ABCDE'>
0420   2 #A34       (A5) INIT <'ABCDE'>
0430   2 #A35       (A5) INIT <'ABCDE'>
0440   2 #A36       (A5) INIT <'ABCDE'>
0450   2 #A37       (A5) INIT <'ABCDE'>
0460   2 #A38       (A5) INIT <'ABCDE'>
0470   2 #A39       (A5) INIT <'ABCDE'>
0480   2 #A40       (A5) INIT <'ABCDE'>
0490   2 #A41       (A5) INIT <'ABCDE'>
0500   2 #A42       (A5) INIT <'ABCDE'>
0510   2 #A43       (A5) INIT <'ABCDE'>
0520   2 #A44       (A5) INIT <'ABCDE'>
0530   2 #A45       (A5) INIT <'ABCDE'>
0540   2 #A46       (A5) INIT <'ABCDE'>
0550   2 #A47       (A5) INIT <'ABCDE'>
0560   2 #A48       (A5) INIT <'ABCDE'>
0570   2 #A49       (A5) INIT <'ABCDE'>
0580   2 #A50       (A5) INIT <'ABCDE'>
0590 1 REDEFINE #GROUP
0600   2 #ARRAY (A5/1:50)
0610 1 #LOOP           (P7)
0620 END-DEFINE
0630 *
0640 INCLUDE AATITLER
0650 INCLUDE AASETC
0660 *
0670 FOR #LOOP = 1 TO 50000
0680 WRITE WORK FILE 1 #GROUP
0690 END-FOR
0700 *
0710 SETA. SETTIME
0720 READ WORK FILE  1  #GROUP
0730 END-WORK
0740 WRITE // 10T 'READ WORK FILE GROUP TIME' *TIMD (SETA.)
0750 *
0760 SETB. SETTIME
0770 READ WORK FILE  1 #ARRAY (*)
0780 END-WORK
0790 WRITE // 10T 'READ WORK FILE ARRAY TIME' *TIMD (SETB.)
0800 *
0810 END
```

And the surprising (to me) output.

```
 PAGE #   1                    DATE:    Dec 16, 2001
PROGRAM: READW07             LIBRARY: INSIDE


     READ WORK FILE GROUP TIME       18


     READ WORK FILE ARRAY TIME       17
```

Wrong again. The times are the same. You will recall that the array notation made a considerable difference for WRITE WORK FILE. The same is not true for READ WORK FILE. It took me a couple of minutes to realize why.

READ WORK FILE RECORD versus READ WORK FILE

It is time (in order to explain the output from READW07) to discuss the major difference between RWF and RWF RECORD. In a very real sense, they were designed to address two different scenarios.

Scenario 1) We are receiving a Work File from one of our remote sites. This site has a history of supplying “bad data”. Numeric fields have alpha data; Alpha fields have “garbage characters”, etc.

Scenario 2) We have a large system. The system has been designed to work as several job steps. In order to “pass data” between the job steps, Work Files are typically written in one job step, then read in a subsequent job step.

The two scenarios above probably constitute 98% of the use of Work Files. Consider the second scenario. Is there a reason, when reading the work-files you just created, to check and ensure every field has a valid value (format and length)? Clearly not. How about the first scenario. Just as clearly, the answer is yes.

READ WORK FILE (without RECORD) addresses scenario 1. It checks the validity of every field. This is quite CPU intensive. By contrast, RWF RECORD addresses scenario 2. There is no value checking. How significant is the performance difference? Take a look at the following program:

```
* THIS PROGRAM CREATES A WORK FILE WITH 50000 RECORDS
*
* THEN WE COMPARE THE TIME TO READ THIS WORK FILE
* USING READ WORK FILE AND READ WORK FILE RECORD.
*
* THE RATIO ON MY SYSTEM IS ABOUT 5 TO 1 IN FAVOR OF
* THE READ WORK FILE RECORD.
*
DEFINE DATA LOCAL
1 #GROUP (50)
  2 #ALPHA        (A5) INIT ALL <'ABCDE'>
  2 #NUMERIC      (N5) INIT ALL <12345>
1 #LOOP           (N5)
END-DEFINE
*
INCLUDE AATITLER
INCLUDE AASETC
*
FOR #LOOP = 1 TO 50000
WRITE WORK FILE 1 #GROUP (*)
END-FOR
*
RWFR. SETTIME
READ WORK FILE  1  RECORD #GROUP (*)
END-WORK
WRITE / 10T 'READ WORK FILE RECORD TIME' *TIMD(RWFR.)
*
RWF. SETTIME
READ WORK FILE  1  #GROUP (*)
END-WORK
WRITE // 10T 'READ WORK FILE TIME' *TIMD (RWF.)
*
END
```

And the rather graphic output:

```
PAGE #   1                    DATE:    01-12-17
PROGRAM: READW04              LIBRARY: SNDEMO


     READ WORK FILE RECORD TIME       7


     READ WORK FILE TIME       33
```

What does this say? Do NOT use READ WORK FILE if READ WORK FILE RECORD will suffice. Now to be fair, lets look at one more program.

```
0010 * THIS PROGRAM CREATES A WORK FILE WITH 150000 RECORDS
0020 * EACH OF WHICH HAS BUT THREE FIELDS.
0030 *
0040 * THEN WE COMPARE THE TIME TO READ THIS WORK FILE
0050 * USING READ WORK FILE AND READ WORK FILE RECORD.
0060 *
0070 DEFINE DATA LOCAL
0080 1 #GROUP
0090   2 #A (A3) INIT <'ABC'>
0100   2 #B (N5) INIT <12345>
0110   2 #C (A5) INIT <'ABCDE'>
0120 1 #LOOP (P7)
0130 END-DEFINE
0140 *
0150 INCLUDE AATITLER
0160 INCLUDE AASETC
0170 *
0180 FOR #LOOP = 1 TO 150000
0190 WRITE WORK FILE 1 #A #B #C
0200 END-FOR
0220 *
0240 RWFR. SETTIME
0250 READ WORK FILE  1  RECORD #GROUP
0260 END-WORK
0270 WRITE // 10T 'READ WORK FILE RECORD TIME'
0280          *TIMD (RWFR.)
0290 *
0300 RWF. SETTIME
0310 READ WORK FILE  1  #GROUP
0320 END-WORK
0330 WRITE // 10T 'READ WORK FILE TIME' *TIMD (RWF.)
0340 *
0350 END
```

And our rather different (from above) output.

```
PAGE #   1                    DATE:    Dec 17, 2001
PROGRAM: READW04              LIBRARY: INSIDE


     READ WORK FILE RECORD TIME       10


     READ WORK FILE TIME       10
```

Note that the performance difference diminishes as the number of fields is reduced. Not surprising. The main difference, as noted above, is validity checking of fields (yes, there also is some difference, to be discussed, about how the data is placed). Reduce the number of fields to a small number, say less than five, and there is basically no difference between the two commands.

What does this all mean?

Notice that for READ WORK FILE, Natural must check the format of all the variables. Thus, #ARRAY (1) #ARRAY (2),..etc. must be checked individually. Hence, Natural cannot simply read in a single “chunk” of #ARRAY (1) thru #ARRAY (50). They are all treated individually, hence the array syntax, #ARRAY (*) is no faster than the Group syntax (see output from READW07, earlier in the article).

So, array versus non array (individual variables) is meaningless for READ WORK FILE. For READ WORK FILE RECORD, there is no comparison to be made since individual variables are not allowed.

Back to RWF RECORD

When Natural first came out, the READ WORK FILE RECORD option required a single variable. That variable could be an array. Somewhere around Version 2.1 or 2.2, Natural was “enhanced”. Group Names suddenly became valid, and along with them, Group arrays. HOWEVER, this is an “illusion”. What do I mean?

Consider the following program:

```
0010 DEFINE DATA LOCAL
0020 1 GROUP-NAME-A
0030   2 #A1 (A3) INIT  <'ABC'>
0040   2 #A2 (A3) INIT  <'123'>
0050   2 #A3 (A3) INIT  <'D5F'>
0060 1 GROUP-NAME-B
0070   2 #B1 (N3)
0080   2 #B2 (A3)
0090   2 #B3 (N3)
0100 1 REDEFINE GROUP-NAME-B
0110   2 #B (A9)
0120 1 #LOOP (P7)
0130 1 #STRING (A200)
0140 END-DEFINE
0150 *
0160 INCLUDE AATITLER
0170 INCLUDE AASETC
0180 *
0190 WRITE WORK FILE 2 GROUP-NAME-A
0200 *
0210 READ WORK FILE 2 ONCE RECORD  GROUP-NAME-B
0220 *
0230 WRITE // 5T '=' #B1 5X '=' #B2 5X '=' #B3
0240       // 5T '=' #B
0250 END
```

The program runs and produces this output.

```
PAGE #   1                    DATE:    Dec 17, 2001
PROGRAM: WORK19               LIBRARY: INSIDE



#B1:  AB3     #B2: 123     #B3:  D56

#B: ABC123D5F
```

This is the only place in Natural where a Group Name is not an abbreviation for its component fields. Basically, Natural simply uses the Group Name to supply a starting address for the RWFR command. Remember that the RECORD option means that field formats are being ignored. You can see that in the output above. Natural read alpha fields into numeric fields without so much as blinking.

How do I know that the individual fields are being ignored? Here is the last program with one minor change.

```
0010 DEFINE DATA LOCAL
0020 1 GROUP-NAME-A
0030   2 #A1 (A3) INIT  <'ABC'>
0040   2 #A2 (A3) INIT  <'123'>
0050   2 #A3 (A3) INIT  <'D5F'>
0060 1 GROUP-NAME-B
0070   2 #B1 (N3)
0080   2 #B2 (A3)
0090   2 #B3 (N3)
0100 1 REDEFINE GROUP-NAME-B
0110   2 #B (A9)
0120 1 #LOOP (P7)
0130 1 #STRING (A200)
0140 END-DEFINE
0150 *
0160 INCLUDE AATITLER
0170 INCLUDE AASETC
0180 *
0190 WRITE WORK FILE 2 GROUP-NAME-A
0200 *
0210 READ WORK FILE 2 ONCE RECORD #B1 #B2 #B3
0220 *
0230 WRITE // 5T '=' #B1 5X '=' #B2 5X '=' #B3
0240       // 5T '=' #B
0250 END
0260
```

Note that the READ WORK FILE command on line 0210 now has three variables. Here is the compiler error message:

NAT0077 Error in data field for READ/WRITE WORK FILE statement.

A “gotcha”

There is one aspect of READ/WRITE WORK FILE that deserves special mention here. It has to do with arrays. To be more precise, it has to do with what I call “Group Arrays”. This is a structure such as:

```
1    #GROUP (10)
     2    #A (A5)
     2    #B (A3)
```

Please recall from somewhere at the beginning of this article, for such a structure, the values are arranged in memory as:

#A (1) #B (1) #A (2) #B (2)……#A (10) #B (10)

Now for the fun.

Suppose I simply had the statement WRITE #GROUP (*). The question is, in what sequence will the data appear? Basically, there are only two alternatives one can imagine:

#A (1) #B (1) #A (2) #B (2)……#A (10) #B (10)
or
#A (1) #A (2)…#A (10) #B (1) #B (2)……#B (10)

In other words, either we get

#GROUP (1) #GROUP (2)…#GROUP(10)
or,
#A (*) #B (*).

The following program and output resolves this question.

```
0010 * THIS PROGRAM DEMONSTRATES THE WRITE'ING OF A GROUP
0020 * ARRAY. IT ALSO DEPICTS THE ARRANGEMENT OF A GROUP
0025 * ARRAY IN MEMORY.
0030 *
0040 * NOTE THAT DATA IS ARRANGED "GROUPWISE" IN MEMORY.
0050 * HOWEVER A WRITE OF #GROUP (*) IS EQUIVALENT TO:
0060 * WRITE #A (*) #B (*) #C (*) WHICH IN TURN IS
0070 * EQUIVALENT TO #A (1) #A (2) #A (3) #B (1) #B (2)
0075 * #B (3) #C (1) #C (2) #C (3)
0080 * WHICH IS WHAT I CALL "FIELD WISE".
0090 *
0100 DEFINE DATA LOCAL
0110 1 #GROUP (3)
0120   2 #A (A3) INIT <'A1A','A2A','A3A'>
0130   2 #B (A3) INIT <'B1B','B2B','B3B'>
0140   2 #C (A3) INIT <'C1C','C2C','C3C'>
0150 1 REDEFINE #GROUP
0160   2 #STRING (A27)
0170 END-DEFINE
0180 *
0190 INCLUDE AATITLER
0200 INCLUDE AASETC
0210 *
0220 WRITE 5T 'OUR DATA STRUCTURE IS:' /
0230       5T '1 #GROUP (3)' /
0240       5T '  2 #A (A3) INIT <"A1A","A2A","A3A">' /
0250       5T '  2 #B (A3) INIT <"B1B","B2B","B3B">' /
0260       5T '  2 #C (A3) INIT <"C1C","C2C","C3C">' /
0270       5T '1 REDEFINE #GROUP' /
0280       5T '  2 #STRING (A27)' //
0290   5T 'THE NEXT LINE SHOWS MEMORY VIA WRITE #STRING' /
0300       5T #STRING //
0310       5T 'THE NEXT LINE SHOWS WRITE #GROUP (*)' /
0320       5T #GROUP (*)
0330 *
0340 END
```

And our output which answers the question posed above.

```
PAGE #   1                    DATE:    01-12-16
PROGRAM: WRITE05              LIBRARY: SNDEMO

OUR DATA STRUCTURE IS:
1 #GROUP (3)
  2 #A (A3) INIT <'A1A','A2A','A3A'>
  2 #B (A3) INIT <'B1B','B2B','B3B'>
  2 #C (A3) INIT <'C1C','C2C','C3C'>
1 REDEFINE #GROUP
  2 #STRING (A27)

THE NEXT LINE SHOWS MEMORY VIA WRITE #STRING
A1AB1BC1CA2AB2BC2CA3AB3BC3C

THE NEXT LINE SHOWS WRITE #GROUP (*)
A1A A2A A3A B1B B2B B3B C1C C2C C3C
```

Okay, question answered. For a WRITE #GROUP(*) (and as we shall see a WRITE WORK FILE) statement, data is written as if the statement was WRITE #A (*) #B (*) #C(*). Hence, the data is written to our screen, “fieldwise”.

Okay, now lets change the WRITE to a WRITE WORK FILE and do a READ WORK FILE into #A (*) #B (*) #C(*) to confirm than the work file was indeed written fieldwise.

```
* THIS PROGRAM DEMONSTRATES THE WRITE'ING OF A
* GROUP ARRAY TO A WORK FILE. ALSO, WE WILL READ THE
* WORK FILE USING THE SAME SYNTAX.
*
DEFINE DATA LOCAL
1 #GROUP (3)
  2 #A (A3) INIT <'A1A','A2A','A3A'>
  2 #B (A3) INIT <'B1B','B2B','B3B'>
  2 #C (A3) INIT <'C1C','C2C','C3C'>
1 REDEFINE #GROUP
  2 #STRING (A27)
END-DEFINE
*
INCLUDE AATITLER
INCLUDE AASETC
*
WRITE 5T
    'DATA BEFORE WRITE AND READ WORK FILE #GROUP(*)'
      /  5T '-' (50)
      /  5T '=' #A (*)
      /  5T '=' #B (*)
      /  5T '=' #C (*)
WRITE WORK FILE 1 #GROUP (*)
READ WORK FILE 1 ONCE #GROUP (*)
WRITE // 5T
     'DATA AFTER WRITE AND READ WORK FILE #GROUP(*)'
      /  5T '-' (50)
      /  5T '=' #A (*)
      /  5T '=' #B (*)
      /  5T '=' #C (*)
*
END
```

And the expected output.

```
PAGE #   1                    DATE:    01-12-16
PROGRAM: WRITE05X             LIBRARY: SNDEMO

DATA BEFORE WRITE AND READ WORK FILE #GROUP(*)
--------------------------------------------------
#A: A1A A2A A3A
#B: B1B B2B B3B
#C: C1C C2C C3C


DATA AFTER WRITE AND READ WORK FILE #GROUP(*)
--------------------------------------------------
#A: A1A A2A A3A
#B: B1B B2B B3B
#C: C1C C2C C3C
```

Note that for the statement READ WORK FILE #GROUP (*) , the #GROUP (*) is interpreted the same for the WRITE WORK FILE #GROUP (*).

Now for the fun. How about READ WORK FILE RECORD #GROUP (*)?

```
* THIS PROGRAM SHOWS THE WRITE'ING OF A GROUP ARRAY
* TO A WORK FILE. ALSO, WE WILL READ THE WORK FILE
* USING THE SAME SYNTAX, BUT IN A READ
* WORK FILE RECORD.
*
DEFINE DATA LOCAL
1 #GROUP (3)
  2 #A (A3) INIT <'A1A','A2A','A3A'>
  2 #B (A3) INIT <'B1B','B2B','B3B'>
  2 #C (A3) INIT <'C1C','C2C','C3C'>
1 REDEFINE #GROUP
  2 #STRING (A27)
END-DEFINE
*
INCLUDE AATITLER
INCLUDE AASETC
*
WRITE  5T
   'DATA BEFORE WRITE AND READ WORK FILE #GROUP(*)'
      /  5T '-' (50)
      /  5T '=' #A (*)
      /  5T '=' #B (*)
      /  5T '=' #C (*)
WRITE WORK FILE 1 #GROUP (*)
READ WORK FILE  1 ONCE RECORD   #GROUP (*)
WRITE // 5T 'DATA AFTER WRITE AND READ WORK
RECORD FILE #GROUP(*)'
      /  5T '-' (50)
      /  5T '=' #A (*)
      /  5T '=' #B (*)
      /  5T '=' #C (*)
*
END
```

And the perhaps unexpected output:

```
PAGE #   1                    DATE:    01-12-18
PROGRAM: WRITE05Y             LIBRARY: SNDEMO

DATA BEFORE WRITE AND READ WORK FILE #GROUP(*)
--------------------------------------------------
#A: A1A A2A A3A
#B: B1B B2B B3B
#C: C1C C2C C3C


DATA AFTER WRITE AND READ WORK FILE RECORD #GROUP(*)
--------------------------------------------------
#A: A1A B1B C1C
#B: A2A B2B C2C
#C: A3A B3B C3C
```

Okay, what happened???

Compare the output above (WRITE05Y) with the last output (WRITE05X). Note the difference in the output.

The explanation is quite simple.

The WRITE WORK FILE #GROUP (*) in both programs is "translated" into:

WRITE #A (*) #B (*) #C(*).

Thus, the workfile looks like:

A1A A2A A3A B1B B2B B3B C1C C2C C3C
(spaces added for readability)

What does READ WORK FILE RECORD #GROUP (*) do?

It does NOT get "translated" to anything. Remember from the discussion above, #GROUP (*) is basically just supplying a starting address for the read. Remember, from much earlier, our discussion of the sequence of occurrences of a group array in memory? It was:

#A(1)#B(1) #C(1) #A(2) #B(2) #C(2) #A(3) #B(3) #C(3)

Try "matching" the actual data with where READ WORK FILE RECORD will put the data (sequential memory locations, starting with the start of #GROUP(*), which is #A(1))

Clearly: A1A => #A(1) A2A => #B(1) A3A => #C(1)
B1B => #A(2) etc.

Hence the rather strange output from WRITE05Y.

Be VERY careful when using Group Arrays with READ WORK FILE RECORD. Unlike READ WORK FILE, where the same syntax as a WRITE WORK FILE will return the data to the original "buckets" from where they came, READ WORK FILE RECORD will alter the contents sequencing.

Is there a "solution" to this problem?

First thing to note; there is no problem, unless you consider understanding how Natural works a problem. I have always considered such knowledge an asset, not a problem.

Now that we know how the READ WORK FILE RECORD works, it is a simple matter to alter our WRITE WORK FILE to be "compatible" with the READ WORK FILE RECORD.

Instead of WRITE WORK FILE #GROUP (*), which Natural "translates" to #A (*) #B (*) #C(*), we could simply WRITE WORK FILE #GROUP (1) #GROUP (2) #GROUP (3); which would put our data in the proper sequence for READ WORK FILE RECORD #GROUP (*).

Alternatively, if we must work with a given work file sequence (whoever wrote that piece of code cannot be influenced to change it), we can alter our READ WORK FILE RECORD structure. We could have:

```
* THIS PROGRAM SHOWS THE WRITE'ING OF A GROUP ARRAY
* TO A WORK FILE. ALSO, WE WILL READ THE WORK FILE
* USING SAME SYNTAX, BUT IN A READ WORK FILE RECORD.
*
DEFINE DATA LOCAL
1 #GROUP (3)
  2 #A (A3) INIT <'A1A','A2A','A3A'>
  2 #B (A3) INIT <'B1B','B2B','B3B'>
  2 #C (A3) INIT <'C1C','C2C','C3C'>
1 #GROUP2
  2 #AREAD (A3/1:3)
  2 #BREAD (A3/1:3)
  2 #CREAD (A3/1:3)
END-DEFINE
*
INCLUDE AATITLER
*
WRITE  5T
 'DATA BEFORE WRITE AND READ WORK FILE #GROUP(*)'
      /  5T '-' (50)
      /  5T '=' #A (*)
      /  5T '=' #B (*)
      /  5T '=' #C (*)
WRITE WORK FILE 1 #GROUP (*)
READ WORK FILE  1 ONCE RECORD   #GROUP2
WRITE // 5T
 'DATA AFTER WRITE AND READ WORK FILE #GROUP(*)'
      /  5T '-' (50)
      /  5T '=' #AREAD (*)
      /  5T '=' #BREAD (*)
      /  5T '=' #CREAD (*)
*
END
```

And the output:

```
PAGE #   1                  DATE:    01-12-18
PROGRAM: WRITE05Z           LIBRARY: SNDEMO

DATA BEFORE WRITE AND READ WORK FILE #GROUP(*)
--------------------------------------------------
#A: A1A A2A A3A
#B: B1B B2B B3B
#C: C1C C2C C3C


DATA AFTER WRITE AND READ WORK FILE #GROUP(*)
--------------------------------------------------
#AREAD: A1A A2A A3A
#BREAD: B1B B2B B3B
#CREAD: C1C C2C C3C
```

As you can see, the arrays #AREAD, #BREAD, and #CREAD are the same as #A, #B, and #C. Basically, we just have to get the WRITE and READs to be compatible. We can do this by changing either the structures (used for READ/WRITE), or changing the WRITE WORK FILE statement. Note we cannot change the READ WORK FILE RECORD structure since it is only a single variable or group.

Summary

Contrary to "common knowledge", neither READ nor WRITE WORK FILE take advantage of variable sequencing in memory.

However, array notation greatly impacts WRITE WORK FILE performance. Also, the number of variables being written greatly impacts performance of WWF.

Array notation does not impact the performance of READ WORK FILE since each occurrence must still be checked for valid format.

READ WORK FILE RECORD greatly outperforms READ WORK FILE, however, one must be careful when using Group Arrays. ❖

And now for something entirely different

Whereas Thomas's topics (try saying that fast three times) were mainly performance oriented, Andreas topics were quite different. The title of Andreas's talk, "Seldom used or unknown features in Natural" is a giveaway to the emphasis of his presentation. Andreas, like Thomas, had over twenty distinct topics and just one hour to present them all. Hence, I will take several of Andreas's topics and expand upon them here.

COMPRESS NUMERIC

Perhaps the most common documentation criticism I have heard over the lifetime of Natural is the use of Release Notes to document new features. In most Adabas/Natural shops, the DBA is the recipient of Release Notes. They use them to install the new release, then place them in a folder somewhere, never to be seen again. Software AG is working on this, and soliciting ideas as to how to get the Release Notes into the hands of the programmers.

One new facility so documented is the new NUMERIC option for the COMPRESS command. This was added with SM 3 (or was it 2) of Version 2.3, yet many, probably most, programmers, are not familiar with it. Here is an example Andreas used to demonstrate precisely how COMPRESS and COMPRESS NUMERIC work.

```
0010 * THIS PROGRAM DEMONSTRATES THE DIFFERENCE BETWEEN
0020 * COMPRESS AND COMPRESS NUMERIC
0030 *
0040 DEFINE DATA LOCAL
0050 1 #A (N3)   INIT <-123>
0060 1 #B (N1.2) INIT <1.23>
0070 1 #OUTPUT (A10)
0080 END-DEFINE
0090 *
0100 INCLUDE AATITLER
0110 INCLUDE AASETC
0120 *
0130 COMPRESS #A #B INTO #OUTPUT WITH DELIMITER '*'
0140 WRITE // 5T 'DATA FOR THIS PROGRAM' / 5T '-' (25)
0150       / 5T '=' #A 5X '=' #B ///
0160       5T 'RESULT OF: COMPRESS #A #B INTO'
0165          '  #OUTPUT WITH DELIMITER "*"'
0170       // 5T '=' #OUTPUT
0180 *
0190 COMPRESS NUMERIC #A #B INTO #OUTPUT WITH DELIMITER '*'
0200 WRITE //
0210       5T 'RESULT OF: COMPRESS NUMERIC #A #B'
0215          'INTO #OUTPUT WITH DELIMITER "*"'
0220       // 5T '=' #OUTPUT
0230 *
0240 END
```

And here is our output:

```
PAGE #   1                   DATE:    Dec 06, 2001
PROGRAM: COMPR01             LIBRARY: INSIDE


DATA FOR THIS PROGRAM
------------------------
#A: -123    #B:  1.23


RESULT OF: COMPRESS #A #B INTO #OUTPUT
           WITH DELIMITER '*'

#OUTPUT: 123*123


RESULT OF: COMPRESS NUMERIC #A #B INTO #OUTPUT
           WITH DELIMITER '*'

#OUTPUT: -123*1.23
```

Okay, it is quite clear what COMPRESS NUMERIC does. It includes both minus signs and decimal points. Why doesn't a simple COMPRESS do this? If you were to write out both #A and #B with hex edit masks (e.g. WRITE #B (EM=HHH)), you would discover that the actual contents of #B is the digits 123. There is no decimal point physically within the confines of #B.

However, Natural "knows" that the value of #B is 1.23, not 123. If you do arithmetic with #B, Natural "knows" where the decimal point is. If you WRITE #B, as you can see in our output, Natural writes out the implied decimal point.

HOWEVER, if you simply COMPRESS with #B, Natural treats #B as if it were an alpha string. As we have discussed, the contents of #B (and #A) is 123. Hence, the perhaps unexpected result of 123*123.

COMPRESS NUMERIC, by contrast, recognizes that #A and #B are indeed numbers. It checks the format of the two variables and inserts appropriate minus sign(s) and decimal points.

My problem with COMPRESS NUMERIC is that it only takes me partway to what I usually require. I do find it useful for things like percentages where I want to create a variable with a value like 84.23 %. However, when I want to create a variable with a value such as $ 12,345.67, COMPRESS NUMERIC will not accomplish my objective. Here is a program which demonstrates both the uses just cited.

```
0010 * THIS PROGRAM DEMONSTRATES THE USE OF COMPRESS
0020 * NUMERIC FOR PERCENTAGES AND MONETARY VALUES.
0030 *
0040 DEFINE DATA LOCAL
0050 1 #A (N8.2)   INIT <12345678.99>
0060 1 #B (N3.2)   INIT <86.25>
0070 1 #OUTPUT (A15)
0080 END-DEFINE
0090 *
0100 INCLUDE AATITLER
0110 INCLUDE AASETC
0120 *
0130 COMPRESS NUMERIC #B ' %' INTO #OUTPUT LEAVING NO SPACE
0140 WRITE 5T 'HERE IS A GOOD USE FOR COMPRESS NUMERIC'
0150     / 5T 'THE VALUE OF #OUTPUT IS:' #OUTPUT
0160 *
0170 COMPRESS NUMERIC '$' #A  INTO #OUTPUT LEAVING NO SPACE
0180 WRITE // 5T 'THIS IS NOT SUCH A GOOD USE FOR'
0185             ' COMPRESS NUMERIC'
0190      / 5T 'RESULT OF: COMPRESS NUMERIC "$ " #A
0195           INTO #OUTPUT LEAVING NO SPACE'
0200       // 5T '=' #OUTPUT
0210 *
0220 END
```

And our output:

```
PAGE #   1                   DATE:    Dec 07, 2001
PROGRAM: COMPR02             LIBRARY: INSIDE

HERE IS A GOOD USE FOR COMPRESS NUMERIC
THE VALUE OF #OUTPUT IS: 86.25 %


THIS IS NOT SUCH A GOOD USE FOR COMPRESS NUMERIC
RESULT OF: COMPRESS NUMERIC '$ ' #A INTO
#OUTPUT LEAVING NO SPACE

#OUTPUT: $12345678.99
```

The problem with COMPRESS NUMERIC for monetary amounts is quite clear. Although we are able to get the dollar sign and decimal point in the desired position (I could have a space after the dollar sign if so desired), I am missing the familiar delimiters which would separate the value into "thousands", namely 12,345,678.99.

There are many ways to address such a requirement, so I will depart from Andreas's presentation to discuss this.

Tangent

There are two distinct scenarios for creating the type of output just described. Each has its own solutions. The first scenario simply involves the desire to DISPLAY such a value, along with others. Note that DISPLAY is in capital letters indicating this is the statement to be employed (as opposed to WRITE, PRINT, etc).

There is a very powerful, yet not widely known (recall the title of Andreas's talk, "Seldom used or unknown features in Natural") combination of parameters that exists for just the DISPLAY statement. Here is an example of this, along with a more conventional, single parameter.

```
0010 * THIS PROGRAM DEMONSTRATES THE USE OF TWO DIFFERENT
0020 * MASKS IN CONJUNCTION WITH DISPLAY
0030 *
0040 DEFINE DATA LOCAL
0050 1 #A (N8.2)   INIT <12345678.99>
0060 END-DEFINE
0070 *
0080 INCLUDE AATITLER
0090 INCLUDE AASETC
0100 *
0110 DISPLAY 5T #A (EM=ZZ,ZZZ,ZZ9.99)
0120        5X #A (EM=ZZ,ZZZ,ZZ9.99 IC=$)
0130 *
0140 END
```

And our output:

```
PAGE #   1                  DATE:    Dec 07, 2001
PROGRAM: COMPR02X           LIBRARY: INSIDE

     #A                #A
-------------     --------------

12,345,678.99     $12,345,678.99
```

Note that the use of the Insert Character, IC=$, allows us to insert the dollar sign. However, suppose the value did not fill the field.

```
0010 * THIS PROGRAM DEMONSTRATES THE USE OF TWO DIFFERENT
0020 * MASKS IN CONJUNCTION WITH DISPLAY
0030 *
0040 DEFINE DATA LOCAL
0050 1 #A (N8.2)   INIT <12345.99>
0060 END-DEFINE
0070 *
0080 INCLUDE AATITLER
0090 INCLUDE AASETC
0100 *
0110 DISPLAY 5T #A (EM=ZZ,ZZZ,ZZ9.99)
0120        5X #A (EM=ZZ,ZZZ,ZZ9.99 IC=$)
0130 *
0140 END
```

And here is our output.

```
PAGE #   1                  DATE:    Dec 07,
PROGRAM: COMPR02Y           LIBRARY: INSIDE

     #A                #A
-------------     --------------

    12,345.99         $12,345.99
```

Not to worry, the Z's in our edit mask cause the elimination of leading zeroes, and the dollar sign becomes a "floating" character.

One more little "goodie" to discuss. The Insert Character does not have to be a single character. Why is this useful? Suppose I did not want the dollar sign to be contiguous with the value, but instead, wanted a blank after the dollar sign. Here is a program which demonstrates a multi character IC.

```
0010 * THIS PROGRAM DEMONSTRATES THE USE OF TWO DIFFERENT
0020 * MASKS IN CONJUNCTION WITH DISPLAY
0030 *
0040 DEFINE DATA LOCAL
0050 1 #A (N8.2)   INIT <12345.99>
0060 END-DEFINE
0070 *
0080 INCLUDE AATITLER
0090 INCLUDE AASETC
0100 *
0110 DISPLAY 5T #A (EM=ZZ,ZZZ,ZZ9.99)
0120        5X #A (EM=ZZ,ZZZ,ZZ9.99 IC='$ ')
0130 *
0140 END
```

And the output; perhaps more readable than the previous output.

```
PAGE #   1                  DATE:    Dec 07, 2001
PROGRAM: COMPR02Z           LIBRARY: INSIDE

     #A                #A
-------------     ---------------

    12,345.99        $ 12,345.99
```

As you can see, the two character IC makes the output a bit more readable.

WARNING. Thus far, since we were concentrating on the use of EM and IC, I have only DISPLAY'ed a single line of output. Take a look at the following output, which is the same as COMPR02Z output except there are several lines of output, with varying values for #A.

```
PAGE #   1                  DATE:    Dec 07,
PROGRAM: COMPR02Z           LIBRARY: INSIDE

     #A                #A
-------------     ---------------

    12,345.99        $ 12,345.99
 1,234,567.99      $ 1,234,567.99
       123.99           $ 123.99
12,345,678.99     $ 12,345,678.99
```

I personally do not like the "effect" of the floating dollar sign in columns of numbers. I prefer either no dollar sign (as in the column to the left), or a fixed dollar sign as the leftmost character. Fortunately, Natural has a feature similar to IC that achieves the fixed dollar sign. The following syntax, with LC rather than IC, achieves this:

#A (EM=ZZ,ZZZ,ZZ9.99 LC='$ ')

And here is the output shown as a new column to the right of the preceding output.

```
PAGE #   1                   DATE:    Dec 07, 2001
PROGRAM: COMPR02Z            LIBRARY: INSIDE

       #A                 #A                 #A
  -------------     ---------------    ---------------

      12,345.99         $ 12,345.99    $     12,345.99
   1,234,567.99      $ 1,234,567.99    $  1,234,567.99
         123.99            $ 123.99    $        123.99
  12,345,678.99     $ 12,345,678.99    $ 12,345,678.99
```

Of the three, I prefer the output without any dollar sign. However, if different lines were to have different currencies, I would prefer the output to the right over the output in the middle. Clearly, this is highly subjective. Talk to your users regarding which output they would prefer. It is equally simple to generate any of the three in Natural.

Continuing on with our tangent, the second scenario is that we require our "result" in a new variable. This as opposed to merely presenting the output on a screen or report. Recall that the COMPRESS NUMERIC command sufficed to get our decimal point and the currency indicator, but failed to get the desired commas to delineate a string longer than three integers.

Please remember our assumptions. We are starting with a numeric field (e.g. N8.2). We want to end up with an alpha field with contents such as $ 12,345.99. Ideally, this string will be left justified in our variable, although we could always, after the fact, do a MOVE LEFT JUSTIFIED command.

There are a number of ways to approach this problem. I have always found it useful, when discussing this with students, to point out that although COMPRESS, all by itself, is useful for adding "edit characters" to the ends of a numeric field, and COMPRESS NUMERIC is useful for adding decimal points, there is no COMPRESS variation that will "embed" characters such as the comma delimiters we wish to separate every three integers. There is, however, a command that is designed to perform this function (among its many functions), namely MOVE EDITED.

```
0010 * THIS PROGRAM DEMONSTRATES A SHORTCOMING OF
0020 * COMPRESS NUMERIC; AND THE "MESSY" PROCEDURE
0030 * NECESSARY TO PRODUCE A SIMPLE RESULT.
0040 *
0050 DEFINE DATA LOCAL
0060 1 #A (N8.2)   INIT <1234.99>
0070 1 #OUTPUT (A15)
0080 END-DEFINE
0090 *
0100 INCLUDE AATITLER
0110 INCLUDE AASETC
0120 *
0130 COMPRESS NUMERIC '$ ' #A  INTO #OUTPUT
0135                           LEAVING NO SPACE
0140 WRITE //
0150       5T 'RESULT OF: COMPRESS NUMERIC "$ " #A'
0155          ' INTO #OUTPUT LEAVING NO SPACE'
0160       // 5T '=' #OUTPUT
0170 *
0180 MOVE EDITED #A (EM=ZZ,ZZZ,ZZ9.99) TO #OUTPUT
0190 MOVE LEFT #OUTPUT TO #OUTPUT
0200 COMPRESS '$' #OUTPUT INTO #OUTPUT
0210 WRITE // 5T 'RESULT OF: MOVE EDITED'
0215             '#A (EM=" $ "ZZ,ZZZ,ZZ9.99) TO #OUTPUT'
0220       // 5T '=' #OUTPUT
0230 END
```

And our output.

```
PAGE #   1                   DATE:    Dec 07, 2001
PROGRAM: COMPR03             LIBRARY: INSIDE


RESULT OF: COMPRESS NUMERIC '$ ' #A INTO #OUTPUT
           LEAVING NO SPACE

 #OUTPUT: $1234.99


 RESULT OF: MOVE EDITED #A (EM=' $ 'ZZ,ZZZ,ZZ9.99)
            TO #OUTPUT

 #OUTPUT: $ 1,234.99
```

As you can see, the COMPRESS NUMERIC came close, but did not get the commas. The only improvement I have come up with to the code shown above is to eliminate the COMPRESS command.

```
0010 * THIS PROGRAM DEMONSTRATES THE DIFFERENCE BETWEEN
0020 * COMPRESS AND COMPRESS NUMERIC
0030 *
0040 DEFINE DATA LOCAL
0050 1 #A (N8.2)   INIT <1234.99>
0060 1 #OUTPUT (A15)
0070 1 REDEFINE #OUTPUT
0080   2 #DOLLAR (A2)
0090   2 #VALUE (A13)
0100 END-DEFINE
0110 *
0150 MOVE '$ ' TO #DOLLAR
0160 MOVE EDITED #A (EM=ZZ,ZZZ,ZZZ.99) TO #VALUE
0170 MOVE LEFT JUSTIFIED #VALUE TO #VALUE
0180 WRITE // 5T 'RESULT OF: MOVE EDITED AND MOVE LEFT'
0190       // 5T '=' #OUTPUT
0200 END
```

Note the use of a REDEFINE to isolate the dollar sign and a blank.

```
PAGE #   1                    DATE:    Dec 07, 2001
PROGRAM: COMPR04              LIBRARY: INSIDE


RESULT OF: MOVE EDITED AND MOVE LEFT

#OUTPUT: $ 1,234.99
```

MOVE EDITED and MOVE de-EDIT

Andreas discussed one topic that has always been near the top of my list of things to change in Natural. The MOVE EDITED command performs two functions that are basically the opposite of one another. It can be used to add editing characters to a variable, or, it can be used to remove editing characters.

How does Natural know which function you wish to perform? The location of the edit mask. A command like:

MOVE EDITED #A (EM=...) TO #B

will apply the specified edit mask (which usually involves adding edit characters) to #A to produce #B.

By contrast:

MOVE EDITED #C TO #D (EM=...)

will remove the edit characters from #C to produce #D.

I have long thought this should be done via another command like MOVE AND DE-EDIT, or something like that. It is very common to find programmers who were taught the “edit” function of MOVE EDIT, but not taught the “de-edit” function. The following program demonstrates both functions.

```
DEFINE DATA LOCAL
1 #A (A10) INIT <'ABC'>
1 #B (A10)
1 #C (A10)
1 #D (A10)
END-DEFINE
*
INCLUDE AATITLER
INCLUDE AASETC
*
WRITE 5T 'INITIAL VALUES:'
    / 5T '=' #A 5X '=' #B '=' #C //
MOVE EDITED #A (EM=X***X++X) TO #B
WRITE 5T 'VALUES AFTER MOVE EDITED #A (EM=X***X++X) TO #B'
    / 5T '=' #A 5X '=' #B '=' #C //
*
MOVE EDITED #B  TO #C (EM=X***X++X)
WRITE 5T 'VALUES AFTER MOVE EDITED #B (EM=X***X++X) TO #C'
    / 5T '=' #A 5X '=' #B '=' #C

END
```

And the output:

```
PAGE #   1                    DATE:    Dec 07, 2001
PROGRAM: EDITED01             LIBRARY: INSIDE

INITIAL VALUES:
#A: ABC            #B:            #C:


VALUES AFTER MOVE EDITED #A (EM=X***X++X) TO #B
#A: ABC            #B: A***B++C   #C:


VALUES AFTER MOVE EDITED #B (EM=X***X++X) TO #C
#A: ABC            #B: A***B++C   #C: ABC
```

It is fairly simple to follow what happened. The first MOVE EDITED command started with #A, which had value ABC. It applied the edit mask EM=X***X++X and placed the result in #B.

The second MOVE EDITED command is what I prefer to call a MOVE AND DE-EDIT command. It started with #B, whose value is A***B++C. It then “removed” the editing characters contained in the edit mask to create #C, whose value, not surprisingly, is identical to #A.

WARNING

As noted above, there are really two distinct capabilities provided by the single command MOVE EDITED. The edit capability (as opposed to the de-edit capability) is rather forgiving with regard to typos/errors. What do I mean by this? Suppose I want the edit mask shown in the preceding program, but accidently typed (EM=X****X++X), which has one asterisk more than what I really wanted. The program would run, albeit with incorrect output, which I would hopefully notice. Suppose I forgot one of the X’s, something like EM=X***++X? Again, the program would run, but the output would be wrong.

By contrast, the de-edit capability is rather unforgiving with regard to typos. Suppose I omitted one asterisk from the edit mask in the second MOVE EDITED of the program above. I would receive the following error message:

NAT1143 Input does not correspond to input edit mask.

The point is that the de-edit option is examining the source field, looking for the specified edit characters. If they are not there, you get an error message. By contrast, the edit option is not examining the source field. It merely places the edit characters in the appropriate place within the source field. You can even have too many (or too few) X's in the edit mask and your program will still run. ❖

The STACK

I thought it was rather interesting that Andreas included material on the Stack in his presentation. I too have found that many programmers do not even know what Natural's Stack is, and even fewer know what they can do with it. Indeed, I have found that many programmers knowledge of the Stack is limited to the fact that somehow a FETCH command places data on the Stack, and you can "read" this data with an INPUT statement. It has been about three years since I have had an article on the Stack. Next issue I will intersperse text and examples from that article with material from Andreas's presentation. ❖

The %C and %Z Commandsı

have long observed that most programmers do ot know about the so called "Terminal Commands". These are commands that start with a ercent sign and are "delivered" to the Natural Run Module. For this reason, I have always called these "Monitor Commands". Another reason for the name change is that these commands can typically also be issued via a SET CONTROL statement within a Natural program, and hence are quite apart from "terminal" activity. Basically, a percent sign followed by any letter of the alphabet does something in Natural. Andreas only had time to discuss a couple of these commands. Over the next several issues, I will endeavor to cover many more of these commands. This issue, I will discuss the ones that Andreas covered.

We will start with the rather simple %C command. You may have noticed, in almost all my sample programs, two INCLUDE statements. One, AATITLER, is nothing more than a WRITE TITLE statement. The other, AASETC, looks like the following:

```
0010 *
0020 AT END OF PAGE
0030    SET CONTROL 'C'
0040    END-ENDPAGE
0050 *
```

Lets talk about this a bit. As I have discussed in earlier issues, I now run most programs on both a mainframe (Version 3.1.4) and a PC (Version 4.1.2; soon to be Version 5). Actually, I do my first runs on the PC, and merely "confirm" that the mainframe output is the same. When I am working on the mainframe, I take screen snaps using software on my PC (from which I am dialing in to the mainframe).

When I am running on the PC there is a minor problem. Natural on the PC has been enhanced to permit the generation of bit mapped pictures as well as text. The text, therefore is generated as bit maps, not characters. Why is this a problem? Sometimes an output page does not fit into the space I have defined in my page layout software (Pagemaker). As text, it is easy to reduce space (vertically and horizontally) between output; as a bit map, this is very messy (playing in a drawing program).

So, by issuing a SET CONTROL 'C' command at the end of every page, the output not only gets sent to my screen (as a bit map), it also gets sent to the program editor area (as text). It is then a simple matter for me to cut and paste the program and output in whatever manner I please.

One thing to reiterate. The %C command is "effective" only for the next page to be sent from the page buffer to the screen. Thus, I have to place it inside an AT END OF PAGE clause so that all pages will be sent to the source area. Any commands in this clause are executed before the transfer takes place.

Try this facility. Add AASETC to a program. Now run the program. After seeing your output, go into the editor. You will see all your output there. By the way, this is not a bad debugging aid. You get to see your output and the code that generated it. Use the editor's split screen facility to look at both at the same time.

Suppose you want to store away the output of a program as a text member, without the program itself. The command %Z clears the source area. So you might issue a SET CONTROL ‘Z’ at the start of a program, then issue SET CONTROL ‘C’ for every page. You will now have all your output in the source area ready for commands like SET TYPE TEXT and SAVE MYTEXT.

There is another rather interesting use for %C. You can write a Natural program which creates a Natural program. Is this hard? No, and yes. It is not hard to demonstrate how this works. Here is a simple program that does this:

```
0010 * THIS PROGRAM WILL CREATE A SIMPLE NATURAL
0020 * PROGRAM IN THE SOURCE AREA. THEN IT WILL
0030 * STACK A COMMAND TO RUN THE CREATED PROGRAM.
0040 *
0050 SET CONTROL 'Z'   /* CLEARS SOURCE AREA
0060 INCLUDE AASETC
0070 WRITE NOTITLE
0075       'WRITE "THIS IS FROM THE CREATED PROGRAM"'
0080 WRITE 'END'
0090 STACK TOP COMMAND 'RUN'
0100 STOP
0110 END
```

Run the program above. The first screen you see will consist of:

```
WRITE "THIS IS FROM THE CREATED PROGRAM"
END                                          (1)
```

Hit enter. Now you will see:

```
Page 1                   11/02/2001        09:22:12

THIS IS FROM THE CREATED PROGRAM             (2)
```

Hit enter again. You will be back in the source program editor. The contents of the editor, however, will not be the program you saw above. Instead, it will be the two lines:

```
WRITE "THIS IS FROM THE CREATED PROGRAM"
END                                          (3)
```

Some commentary. The first executable statement in the program is SET CONTROL ‘Z’. This clears the program editor area.

Note that I had an INCLUDE AASETC (see discussion above) next in the program. This is a non procedural clause. It does not get executed now.

The next statements are both WRITE statements, the first of which specifies NOTITLE. Together, these statements generate two lines of output in our buffer.

Next, we place a RUN command at the top of our STACK (see preceding article on the use of the Stack).

Next we have a STOP command, which actually starts a lot of things. Since the program is ending, Natural checks the print buffer. There is output waiting to be sent to the screen. Natural now executes the AT END OF PAGE clause which sends the print buffer to the program editor area.

Natural also sends our print buffer to the screen. Hence, we see our output (1) and the program pauses waiting for our action. We hit enter. Control returns to the Natural monitor which “sees” the RUN command waiting for it on the Stack. Natural RUN’s the program in the program editor which consists of a WRITE statement and an END statement.

The program creates the output shown as (2). Note that we do not have a NOTITLE in the WRITE statement, hence we get the page number, date and time. We hit enter. We are back in the program editor. The code sitting there (3), is the code we “created”.

Tangent again

I must confess that I would not use this technique to create a program. I favor using a “skeleton” program with lots of “ampersand variables” that are substituted for at compile time. For those of you who have not seen this technique, here is a simple example.

First, the skeleton program. You cannot STOW this program, only save it. The “ampersand variables” &FILE and &FIELD will be substituted for, by the compiler, with the values of the global variables +FILE and +FIELD.

```
0010 * THIS IS A "SKELETON" PROGRAM
0015 * WITH “AMPERSAND VARIABLES”.
0020 * THERE WILL BE A "DRIVER" PROGRAM WHICH WILL
0030 * INTERACT WITH THE USER; ACQUIRE VALUES TO BE
0040 * SUBSTITUTED IN THIS PROGRAM; AND FINALLY
0050 * RUN THIS PROGRAM
0060 *
0070 HISTOGRAM &FILE &FIELD
0080 DISPLAY &FIELD *NUMBER
0090 LOOP
0100 END
```

And how do you get values in +FILE and +FIELD? Easy. You have a “driver” program like:

```
0010 * THIS IS THE "DRIVER" PROGRAM FOR AMPERS01.
0020 * IT WILL ACQUIRE VALUES FOR +FILE AND +FIELD,
0030 * THEN RUN AMPERS01.
0040 *
0050 INPUT (AD=M) 2/10 'ENTER FILE HERE===>' +FILE (A30)
0060       5/10 'ENTER FIELD HERE==>' +FIELD (A30)
0070 *
0080 RUN 'AMPERS01'
0090 END
```

Note that the driver program RUN’s and does not FETCH AMPERS01. Okay, what happens when AMPERS01 is RUN? Natural takes the contents of the similarly named global variables and substitutes them for the “ampersand variables”. Thus, suppose when the INPUT statement were run, I had typed VEHICLES for the file name and MAKE for the field name. They are now the contents of +FILE and +FIELD.

Natural now does the appropriate substitution of the value of +FILE for &FILE in the HISTOGRAM statement. The value of +FIELD is placed in the two statements that have &FIELD (HISTOGRAM and DISPLAY). Thus AMPERS01 is now the following program:

```
0070 HISTOGRAM VEHICLES MAKE
0080 DISPLAY MAKE *NUMBER
0090 LOOP
0100 END
```

Many shops do not permit the use of this facility. Why? I haven’t a clue. It is the easiest and fastest way to create a system that will permit endusers to generate their own reports. Many shops object to the fact that the “skeleton” programs are source rather than object code. Thus, they have to be compiled. This goes against the idea that “production” environments should contain only object code.

My “standard” reporting skeleton looks like this:

FIND &FILE WITH &WITH
DISPLAY &FIELDS
LOOP
END

More about this skeleton later, first, some hints about creating a user friendly interface to even this simple reporting program (or the preceding skeleton), and, expanding the system to create a fairly sophisticated user tool.

Data Entry

If at all possible, never let the user type anything directly. Why? Consider the HISTOGRAM driver program shown above. Suppose the user makes a typo when entering the file name; they type VEHICLSE instead of VEHICLES. If you do not “catch” this error in the driver program, they will get a compiler error when the driver program RUN’s the skeleton program.

There are two ways to avoid such a disaster. The one I prefer is not to let the user type anything. Instead, I would show the user a list of files and let them select a file name from a list. Then, I would access my Natural file definitions in Predict to select all the appropriate fields (using ADABAS-DE-TYPE) to show the user another list. This list would have all the descriptors from the selected file. Again, the user would select an entry from a list; they would not type the field name.

It should be noted that there are drawbacks to this “list driven” approach. You may find yourself writing different “driver programs” for different users (or user groups). A small price to pay given the alternative. That is, to let the user type things like file names, then validate them and respond with REINPUT if the entry is incorrect. Much messier to code, less efficient, and not nearly as user friendly. This is especially true when expanding the system as will be discussed below.

Building on the FIND model

kay, time to play with the simple FIND model hown above. Here is the first enhancement:

```
FIND &FILE WITH &WITH
&WHERE
DISPLAY &FIELDS
LOOP
END
```

There is a significant difference between the implementation of a WITH clause (which is required for a FIND) and a WHERE clause, which is optional. Note that the word “WITH” appears in our FIND command, but the word WHERE does not.

Why? The user may not select criteria for a WHERE clause. Suppose I had coded the report program with a statement such as:

FIND &FILE WITH &WITH WHERE &WHERE

The Natural compiler would object to the word WHERE if +WHERE were blank. This way, +WHERE either has a complete WHERE clause, or is blank. Since Natural is "freeform", it does not object to a blank line in a program.

Please note; what I will now show you is NOT the way I do this. More about that below. But suppose I had a line in the INPUT statement of the driver such as:

10T 'ENTER WHERE CLAUSE HERE==> ' #WHERE

After the INPUT statement I might have something like:

```
IF #WHERE EQ ' '
  RESET +WHERE
   ELSE
  COMPRESS 'WHERE' #WHERE INTO +WHERE
   END-IF
```

Why would I not handle the input this way? Imagine all the mistakes an enduser might make in typing a value for #WHERE. I would instead guide the user with screens of field names, operators (EQ GE LT, etc) , and constants or field names for the last argument. I would make sure that an alpha constant had apostrophes around it; I do not want to leave that to the enduser. By contrast, however, note that the interface above is perfectly suitable for a programmer. I use this myself all the time. If I make a typo, I get my compiler error message, rerun the driver, and fix my typing. An enduser would not be happy with such an interface.

Okay, time to enhance our system a bit. Users like to see page headers and/or trailers. Easily done. Simply have &WRITETITLE and &WRITETRAILER somewhere in the report program. Somewhere in the INPUT statement have:

12/10 'ENTER HEADER HERE==>' #HEADER

After the INPUT statement:

```
IF #HEADER EQ ' '
         RESET +WRITETITLE
         ELSE
         COMPRESS 'WRITE TITLE' '"' #HEADER '"'
         INTO +WRITETITLE
         END-IF
```

Basically, every additional capability you might want to have adds just one more ampersand variable to the report program. In the driver program, you will need the code to create a text string value for the corresponding Global Variable. I have built programs like this that have optional SORT BY capability, AT BREAK clauses, AT END OF PAGE clauses, AT END OF DATA clauses, COMPUTE statements, ad nauseam.

Making the model into a "system"

At the end of your driver program have code like:

```
STACK TOP COMMAND 'EXEC DRIVER'
RUN 'REPORTER'
```

In this way, when your report is finished, you will be returned to the driver program.

Depending on how elaborate the model is, you might want to show the user an INPUT screen with all the data from their last report. This will make it easy for them to create a report and then, say, run the report for several different cities by only changing the argument of a FIND.

By the way, there is nothing "magical" or required about there being a FIND statement in the report program. This could just as easily be a READ LOGICAL where the user supplies the file and field names and, optionally, starting and ending values.

The best part of all this is, "it isn't hard". Actually, the hardest part is building the driver program. Okay, depending on your shop, it may be convincing someone that it is okay to have source code in a production environment. ❖

Performance

Some of Andreas's hints fell into the performance area, the title of Thomas's talk. Here is a rather simple technique for saving a few machine cycles.

```
0010 * THIS PROGRAM CONTRASTS TWO MOVES
0020 DEFINE DATA LOCAL
0030 1 #A (A3)
0040 1 #LOOP (P7)
0050 END-DEFINE
0060 *
0070 INCLUDE AATITLER
0080 INCLUDE AASETC
```

```
0090 *
0100 SETA. SETTIME
0110 FOR #LOOP = 1 TO 150000
0120 MOVE 'A' TO #A
0130 END-FOR
0140 WRITE 5T 'DIFFERENT SIZE TIME' *TIMD (SETA.)
0150 *
0160 SETB. SETTIME
0170 FOR #LOOP = 1 TO 150000
0180 MOVE 'A  ' TO #A
0190 END-FOR
0200 WRITE // 5T 'SAME SIZE TIME' *TIMD (SETB.)
0210 *
0220 SETC. SETTIME
0230 FOR #LOOP = 1 TO 150000
0240 IGNORE
0250 END-FOR
0260 WRITE // 5T 'FOR LOOP TIME' *TIMD (SETC.)
0270 *
0280 END
```

The difference between the two loops is quite simple. In the second loop, our alpha constant is the same length as the target of the MOVE. Only one operation is required.

In the first loop, our alpha constant is but one character in length. Natural performs two operations; first it nulls out #A, then it moves the constant into the first position of #A.

```
PAGE #   1                   DATE:    Dec 12, 2001
PROGRAM: SIZE01              LIBRARY: SYSTEM

DIFFERENT SIZE TIME       10


SAME SIZE TIME        9


FOR LOOP TIME        8
```

The performance difference reflects our discussion above. When you subtract out the common FOR loop overhead of 8 from both times, the differently sized alpha constant produces a time that is twice the same sized alpha constant; two operations rather than one.

Just for the “fun of it”, I ran the same program on the mainframe (above times are Natural 4 under NT). The results were similar.

```
PAGE #    1                  DATE:    12/13/01
PROGRAM: STEVE302            LIBRARY: SYSTEM

DIFFERENT SIZE TIME        6
SAME SIZE TIME        5
FOR LOOP  TIME        4
```

Actually, I did not immediately run the mainframe version. I was continuing on with my tests on the PC. After confirming Andreas comments regarding alpha constants (on the PC), I decided to confirm the similar comments regarding numeric constants. (Note: Andreas made it a point to remind everyone that numeric constants like 1.00 are stored as packed decimal numbers.) Remember, I was on the PC when I ran the following program:

```
0010 *
0020 DEFINE DATA LOCAL
0030 1 #A (P8.2)
0040 1 #LOOP (P7)
0050 END-DEFINE
0060 *
0070 INCLUDE AATITLER
0080 INCLUDE AASETC
0090 *
0100 SETA. SETTIME
0110 FOR #LOOP = 1 TO 250000
0120 ADD 1 TO #A
0130 END-FOR
0140 WRITE 5T 'DIFFERENT SIZE TIME' *TIMD (SETA.)
0150 *
0160 RESET #A
0170 *
0180 SETB. SETTIME
0190 FOR #LOOP = 1 TO 250000
0200 ADD 1.00 TO #A
0210 END-FOR
0220 WRITE // 5T 'SAME SIZE TIME' *TIMD (SETB.)
0230 *
0240 SETC. SETTIME
0250 FOR #LOOP = 1 TO 250000
0260 IGNORE
0270 END-FOR
0280 WRITE // 5T 'FOR LOOP TIME' *TIMD (SETC.)
0290 *
0300 END
```

As you can see, this is just a numeric counterpart to the preceding example. The point is that there should be a performance difference due to the different formats in the first loop, as opposed to the same formats in the second loop.

```
PAGE #   1                   DATE:    Dec 13, 2001
PROGRAM: SIZE02              LIBRARY: INSIDE

DIFFERENT SIZE TIME       20

SAME SIZE TIME       26

FOR LOOP TIME       14
```

Whoops. The differently sized ADD produced the faster time. Actually, by quite a bit. Subtract out the common 14, and the times are 6 versus 12; a ratio of two to one.

I was doing this at night, when my brain cells function at less than peak capacity. That's my story, and I am sticking to it. Perhaps you already know the problem. I did not at the time. Hence, this is where I switched to the mainframe. First, I ran the alpha program (STEVE300, above), which confirmed the PC results. Then, I ran the numeric counterpart. Here are the results (note; the times below were actually the same for both P8.2 and N8.2 as the format for #A):

```
PAGE #    1                   DATE:    12/13/01
PROGRAM: STEVE300             LIBRARY: SYSTEM

DIFFERENT SIZE TIME        9

SAME SIZE TIME       6

FOR LOOP  TIME       4
```

Now that is what I had expected. Subtracting out the common 4, the SAME SIZE loop outperformed the DIFFERENT SIZE loop by 2.5 to 1.

Why the discrepancy between the PC and main-frame times?

In the vernacular of today, duuuh. Mainframe native arithmetic is packed decimal. PC native arithmetic is integer. Despite the late hour, I realized what I had done, and set out to rectify the problem.

Did I mention it was late at night? Here is my rewrite (I was still on the mainframe):

```
0010 *
0020 DEFINE DATA LOCAL
0030 1 #A (I4)
0040 1 #LOOP (P7)
0050 END-DEFINE
0060 *
0070 INCLUDE AATITLER
0080 INCLUDE AASETC
0090 *
0100 SETA. SETTIME
0110 FOR #LOOP = 1 TO 250000
0120 ADD 1 TO #A
0130 END-FOR
0140 WRITE 5T 'DIFFERENT SIZE TIME' *TIMD (SETA.)
0150 *
0160 RESET #A
0170 *
0180 SETB. SETTIME
0190 FOR #LOOP = 1 TO 250000
0200 ADD 1.00 TO #A
0210 END-FOR
0220 WRITE // 5T 'SAME SIZE TIME' *TIMD (SETB.)
0230 *
0240 SETC. SETTIME
0250 FOR #LOOP = 1 TO 250000
0260 IGNORE
0270 END-FOR
0280 WRITE // 5T 'FOR LOOP TIME' *TIMD (SETC.)
0290 *
0300 END
```

And the output:

```
MORE
    PAGE #    1                   DATE:    12/13/01
    PROGRAM: STEVE303             LIBRARY: SYSTEM

    DIFFERENT SIZE TIME        7

    SAME SIZE TIME      12

    FOR LOOP  TIME       4
```

Whoops again. Why the disparity? Why is "same size" again so much greater than "different size"? Simple. Take a look at the ADD 1.00 to #A. Lets see, this is "mixed mode" arithmetic. Natural has to convert the integer #A to packed decimal, add the packed decimal constant 1.00 to #A, then store the result back in #A.

A switch in the program (see below):

```
>                  > +  Program     STEVE304 Lib SYSTEM
  0010 DEFINE DATA LOCAL
  0020  1 #A (I4)
  0030  1 #ONE (I4) INIT <1>
  0040  1 #LOOP (P7)
  0050  END-DEFINE
  0060 **
  0070  INCLUDE AATITLER
  0080 **
  0090  SETA. SETTIME
  0100  FOR #LOOP = 1 TO 250000
  0110  ADD 1 TO #A
  0120  END-FOR
  0130  WRITE 5T 'DIFFERENT FORMAT TIME' *TIMD (SETA.)
  0140 **
  0150  RESET #A
  0160 **
  0170  SETB. SETTIME
  0180  FOR #LOOP = 1 TO 250000
  0190  ADD #ONE TO #A
  0200  END-FOR
       ::::::
```

And our output:

```
MORE
    PAGE #    1                   DATE:    12/13/01
    PROGRAM: STEVE304             LIBRARY: SYSTEM
    DIFFERENT FORMAT TIME        7
    SAME FORMAT TIME       6
    FOR LOOP  TIME       4
```

This now made sense. All INTEGER format operands outperformed the mixed format add by 50 % (don't forget to subtract the FOR loop time).

Okay, back to the PC for a more meaningful comparison.

```
0010 *
0020 DEFINE DATA LOCAL
0030 1 #A (P8.2)
0040 1 #AONE (P8.2) INIT <1.00>
0050 1 #B (I4)
0060 1 #BONE (I4) INIT <1>
0070 1 #LOOP (P7)
0080 END-DEFINE
0090 *
0100 INCLUDE AATITLER
0110 INCLUDE AASETC
0120 *
0130 SETA. SETTIME
0140 FOR #LOOP = 1 TO 250000
0150 ADD #AONE TO #A
0160 END-FOR
0170 WRITE 5T 'PACKED TIME' *TIMD (SETA.)
0180 *
0190 RESET #A
0200 *
0210 SETB. SETTIME
0220 FOR #LOOP = 1 TO 250000
0230 ADD #BONE TO #B
0240 END-FOR
0250 WRITE // 5T 'INTEGER TIME' *TIMD (SETB.)
0260 *
0270 SETC. SETTIME
0280 FOR #LOOP = 1 TO 250000
0290 IGNORE
0300 END-FOR
0310 WRITE // 5T 'FOR LOOP TIME' *TIMD (SETC.)
0320 *
0330 END
```

```
PAGE #   1                    DATE:    Dec 13, 2001
PROGRAM: SIZE03               LIBRARY: INSIDE

PACKED TIME       27


INTEGER TIME       19


FOR LOOP TIME       13
```

Well that certainly explains the strange results we saw much earlier (SIZE02). Note how much more efficient the integer arithmetic is. In SIZE02, the simulation of a Packed 1.00 is apparently much more expensive than the simulation of a Packed 1. Hence, the strange results from SIZE02.

WARNING

There have been several postings on SAG-L recently from people considering a transition from a mainframe to a PC server. Most of the questions/answers had to do with fairly serious considerations, like different sort sequences (ASCII versus EBCDIC), starting values for supers, etc. There was little discussion of performance. As I mentioned last issue, I will be doing some research in this area. It does appear that standalone PC's can produce performance to rival mainframes that are serving many users.

One area, however, that requires no further research is arithmetic. As noted in the example above, and earlier examples, PC arithmetic is best done as integer arithmetic, mainframe arithmetic is best done as packed decimal. In both cases, same length is important.

Array Subscripts

Natural uses integers for array subscripts. Thus I format subscripts are more efficient than any other format. Here is a program which contrasts I and P formatted subscripts.

```
0010 DEFINE DATA LOCAL
0020 1 #INT1 (I4) INIT <1>
0030 1 #INT2 (I4) INIT <2>
0040 1 #DEC1 (P3) INIT <1>
0050 1 #DEC2 (P3) INIT <2>
0060 1 #ARRAY (A5/1:2)
0070 1 #LOOP (P7)
0080 END-DEFINE
0090 *
0100 INCLUDE AATITLER
0110 INCLUDE AASETC
0120 *
0130 SETA. SETTIME
0140 FOR #LOOP = 1 TO 300000
0150 MOVE #ARRAY (#INT1) TO #ARRAY (#INT2)
0160 END-FOR
0170 WRITE 10T 'INTEGER TIME' *TIMD (SETA.)
0180 *
0190 SETB. SETTIME
0200 FOR #LOOP = 1 TO 300000
0210 MOVE #ARRAY (#DEC1) TO #ARRAY (#DEC2)
0220 END-FOR
0230 WRITE 10T 'DECIMAL TIME' *TIMD (SETB.)
0240 *
0250 SETC. SETTIME
0260 FOR #LOOP = 1 TO 300000
0270 IGNORE
0280 END-FOR
0290 WRITE 10T 'FOR LOOP TIME' *TIMD (SETC.)
0300 *
0310
0320 END
```

And our output.

```
PAGE #   1                    DATE:    Dec 13, 2001
PROGRAM: ARRAY01              LIBRARY: INSIDE

     INTEGER TIME       28
     DECIMAL TIME       29
     FOR LOOP TIME       16
```

Okay, the difference is not all that horrendous. Twelve versus thirteen (after subtracting out the common 16 FOR loop time). It is, however, a very simple habit to get into, and it will save a few CPU cycles.

More performance

I am known as being "anti-Construct". Guilty as charged. One of my complaints with Construct has very little to do with Construct itself. Okay, a little. Many programmers read Construct code, and, for some strange reason, think the code they see is "really good code". Why they would think that is beyond me. Compilers, like Cobol, Fortran, and even Natural, cannot produce as good code as a good assembler programmer. Higher level code generators, like Construct, cannot produce as good code as a good compiler level programmer. These statements, in today's world anyway, are not open to debate. One does not use Construct if performance is a major issue. Yet people look at Construct code and say, "Construct does it this way, so it must be good".

One thing Construct does is create "small objects". Programmers see this and write their own code this way, with unnecessary internal/external subroutines. Is this expensive? Look below:

```
0010 * THIS PROGRAM DEMONSTRATES THE EFFICIENCY
0020 * OF INTERNAL AND EXTERNAL SUBROUTINES IN NATURAL.
0030 * ALSO COMPARED, A CALLNAT TO A SUBPROGRAM.
0040 DEFINE DATA LOCAL
0050 1 #A (A5)
0060 1 #B (A5)
0070 1 #LOOP (P5)
0080 END-DEFINE
0090 *
0130 CTRL. SETTIME
0140 FOR #LOOP = 1 TO 50000
0150 IGNORE
0160 END-FOR
0170 WRITE 3/10 'CONTROL TIME' *TIMD (CTRL.)
0180 *
0190 INLN. SETTIME
0200 FOR #LOOP = 1 TO 50000
0210 MOVE #A TO #B
0220 END-FOR
0230 WRITE // 10T 'INLINE TIME' *TIMD (INLN.)
0240 *
0250 ISUB. SETTIME
0260 FOR #LOOP = 1 TO 50000
0270 PERFORM MOVER
0280 END-FOR
0290 WRITE // 10T 'INTERNAL PERFORM TIME' *TIMD (ISUB.)
0300 *
0310 DEFINE SUBROUTINE MOVER
0320 MOVE #A TO #B
0330 END-SUBROUTINE
0340 *
0350 ESUB. SETTIME
0360 FOR #LOOP = 1 TO 50000
0370 PERFORM EXMOVE #A #B
0380 END-FOR
0390 WRITE // 10T 'EXTERNAL PERFORM TIME' *TIMD (ESUB.)
0400 *
0410 CSUB. SETTIME
0420 FOR #LOOP = 1 TO 50000
0430 CALLNAT 'DEFINS11' #A #B
0440 END-FOR
0450 WRITE // 10T 'CALLNAT TIME' *TIMD (CSUB.)
0460 *
0470 END
```

And the rather enlightening output.

```
PAGE #   1                     DATE:    01-12-13
PROGRAM: DEFINS10              LIBRARY: SNDEMO

     CONTROL TIME       2
     INLINE TIME       3
     INTERNAL PERFORM TIME       4
     EXTERNAL PERFORM TIME      22
     CALLNAT TIME      27
```

Yes, this is a rather absurd piece of code (a simple MOVE) to place in a separate object. Trust me, I have seen isolated code almost as simple. The most ridiculous? An IF statement that did a MOVE to one of two places.

Look at the numbers. An internal subroutine is twice as expensive as inline code. An external subroutine is TEN times as expensive as an internal subroutine, and hence, TWENTY times as expensive as inline code. I am really not sure why the subprogram is so much more expensive than the external subroutine (a future article perhaps).

Now please, do not tell other programmers that Steve Robinson does not approve of subroutines and subprograms. Nothing could be further from the truth. I use them all the time. I just do not use them without some thought as to what I will gain, and what it will cost me.

Remember, every time you call (PERFORM, CALLNAT, FETCH) another object, Natural has to:

Find it in the buffer pool (and load it, if not there)
Deallocate the current object
Allocate the new object; then Call it

When the object ends, control passes back to the initiating object, and the steps above are repeated.

I have seen internal subroutines that are performed from exactly ONE place. In one case the programmer said it improved readability. Sorry, I can enclose code within two lines of fifty asterisks each, with comments, and it will be as readable as any subroutine. The "cost" of this nonsense? Substantial. It was in a READ loop of some two million records. At least it was an internal subroutine. I have seen external subroutines used which are only PERFORM'ed in one program.

Subroutines (in the generic, not Natural sense) have never been efficiency tools. You have always paid a price for them. In return you get simplified maintenance and widespread use. If the latter will not apply to your intended use, you should probably have the code inline. ❖